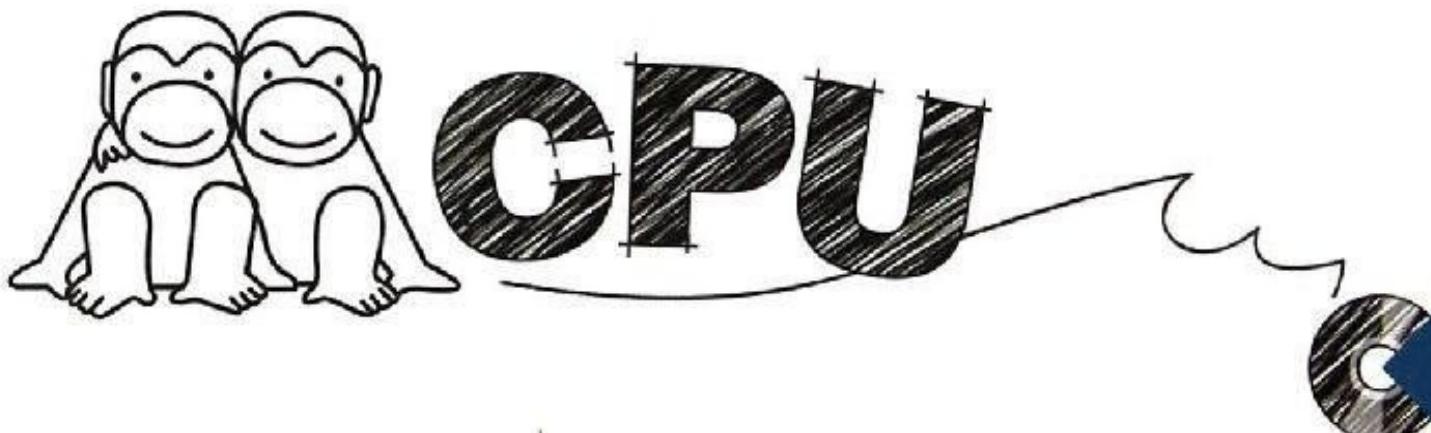


自己动手写 CPU

流水线、指令集、总线……阅读本书，那都不是事儿！

本书手把手地教您从零实现一个具有五级流水线、兼容MIPS32体系结构、带有Wishbone总线接口的处理器，并为其移植了嵌入式操作系统μC/OS-II，打造属于您自己的独一无二的计算机系统。

雷思磊 著



自己动手写 CPU

雷思磊 著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

本书使用Verilog HDL设计实现了一款兼容MIPS32指令集架构的处理器——OpenMIPS。OpenMIPS处理器具有两个版本，分别是教学版和实践版。教学版的主要设计思想是尽量简单，处理器的运行情况比较理想化，与教科书相似，便于使用其进行教学、学术研究和讨论，也有助于学生理解课堂上讲授的知识。实践版的设计目标是能完成特定功能，发挥实际作用。

全书分为三篇。第一篇是理论篇，介绍了指令集架构、Verilog HDL的相关知识。第二篇是基础篇，采用增量模型，实现了教学版OpenMIPS处理器。首先实现了仅能执行一条指令的处理器，从这个最简单的情况出发，通过依次添加，实现逻辑操作指令、移位操作指令、空指令、移动操作指令、算术操作指令、转移指令、加载存储指令、协处理器访问指令、异常相关指令，最终实现了教学版OpenMIPS处理器。第三篇是进阶篇，通过为教学版OpenMIPS添加Wishbone总线接口，从而实现了实践版OpenMIPS处理器，并与SDRAM控制器、GPIO模块、Flash控制器、UART控制器、Wishbone总线互联矩阵等模块组成一个小型SOPC，然后下载到FPGA芯片以验证实现效果，最后为实践版OpenMIPS处理器移植了嵌入式实时操作系统μC/OS-II。

本书适合计算机专业的学生、FPGA开发人员、处理器设计者、嵌入式系统应用开发工程师、MIPS平台开发人员以及对处理器内部的实现感兴趣的读者阅读，也可以作为高等院校计算机原理、计算机体系结构等课程的实践参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

自己动手写CPU／雷思磊著. —北京：电子工业出版社，2014.9

ISBN 978-7-121-23950-2

I . ①自 ... II . ①雷 ... III . ①微处理器－系统设计 IV .
①TP332

中国版本图书馆CIP数据核字（2014）第173014号

策划编辑：孙学瑛

责任编辑：徐津平

特约编辑：顾慧芳

印 刷：北京中新伟业印刷有限公司

装 订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编100036

开 本：787×980 1/16

印 张：34.75

字 数：823千字

版 次：2014年9月第1版

印 次：2014年9月第1次印刷

印 数：3000册

定 价：99.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn， 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

自己动手写处理器?

自己动手写处理器!

没错，您手上拿着的就是一本介绍如何实现处理器的书，通过阅读本书，您可以实现世界上独一无二、独属于您的处理器。

吹牛？

噢，No，理工科学生不打诳语。

不信？

.....那就请您阅读本书。

写作背景

自1971年世界上第一款单芯片微处理器4004诞生已逾40多年，使用“日新月异”来形容这40多年处理器的发展变化亦不为过，无论是速度、集成度，还是架构等许多方面都有了前人难以想象的变化。不过可惜的是，处理器设计制造一直都是高科技行业，轻易无法涉足。大多数人对处理器的直观印象就是一个银白色的小芯片，有许多管脚，至于里面是如何工作的，则不甚了解，更不用说自己制作处理器了。

幸运的是，在处理器发展的同时，可编程逻辑器件也在持续发展。可编程逻辑器件不仅是技术的革新，也带来了观念的革新、设计流程的革新。如今通过编写代码可以在可编程逻辑器件上实现十分复杂的电路设计，比如处理器。于是，普通大众也能有机会了解处理器内部的实现原理，甚至参与处理器的设计、研发。

实际上，目前已经有很多可以下载到可编程逻辑器件上运行的处理器，这些处理器称为软核处理器，比如：NiosII、OR1200、LEON3、OpenSparc等，这些软核处理器有的是开源的，有的不是开源的。笔者在前期深入阅读了几款开源软核处理器的代码，包括：OC8051、OR1200、LEON3。其中，OC8051是OpenCores提供的一款8位两级流水线处理器，与Intel 8051兼容，是CISC（Complex Instruction Set Computer）类型，采用Verilog HDL编写代码。OR1200是OpenCores提供的一款32位五级流水线处理器，是RISC（Reduced Instruction Set Computer）类型，也采用Verilog HDL编写代码。LEON3是由Gaisler Research公司设计发布的一款32位七级流水线处理器，也是RISC类型，但采用的是VHDL编写代码。

通过阅读上述开源软核处理器的代码，一方面消除了笔者对处理器的神秘印象，另一方面也激发了笔者强烈的创作冲动，陆游曾言：纸上得来终觉浅，绝知此事要躬行。是啊！为何不自己也写一个处理器？于是世间又多了一款开源处理器OpenMIPS。OpenMIPS是具有哈佛结构的32位五级流水线标量处理器，兼容MIPS32体系结构，这样可以使用现有的MIPS编译开发环境。它分为教学版和实践版两个版本，每个版本都使用VHDL和Verilog HDL两种语言编写。

本书以Verilog HDL编写的版本为例，详细介绍了OpenMIPS从无到有、从小到大、一步一步成长完善的过程。

写作目的

- 撕掉处理器贴着的“高大上”标签

处理器一贯被人们贴上“高大上”、“高科技”诸如此类的标签，贴标签的原因就在于人们对这些不了解，越不了解，越觉得神秘，越觉得神秘，就越不想了解，如此，“高大上”的标签算是贴牢了。本书的目的之一就是帮助读者改变这种固有的偏见，从本质上认识处理器，撕掉这些标签。简单来说，处理器的作用就是识别0、1编码，也就是识别指令，据此进行各种运算和数据处理。处理器只能计算小学数学课堂上讲授的四则运算，再加上一些并不复杂的与、或、非等逻辑运算，其余诸如平方、开方、微分、积分等都是做不了的。是不是很简单？

- 对现有处理器相关书籍的补充

翻看现有的处理器相关书籍，会有两个体会：一个体会是大多数书籍讲授的理论部分太多，实践部分太少。过多的理论、过少的实践，会使得读者知其然，不知其所以然；第二个体会是有少部分书籍讲授处理器的实现，但是介绍的方式不太易懂，读者需要对处理器有一定了解之后才能阅读此类书籍。因此，笔者想结合OpenMIPS介绍处理器实现，本书完全按照OpenMIPS的实现过程讲解，从零起步，遇到什么问题就解决什么问题，笔者认为这样易于理解。

- 抛砖引玉

高手在民间，此言不虚，笔者写作此书的第三个目的就是抛砖引玉，希望有更多人士能够参与维护和改进OpenMIPS，为其添加更多的

功能，或者改善性能。当然，也希望出现更多类似的软核处理器，这样，大家可以相互学习、相互探讨、取长补短、共同进步。

适合谁读

适合对处理器内部的实现有着强烈好奇心的朋友阅读，通过本书介绍的OpenMIPS处理器的实现过程，您将全方位地了解32位RISC处理器的内部设计。

适合不满足于教科书的同学阅读，本书可以作为您的实践参考书，帮助您理解书本上抽象的概念，同时培养动手能力。

适合正在从事软核处理器开发、设计的朋友阅读，本书将给您一些经验、一些好的方法，帮助您事半功倍。

适合正在从事嵌入式开发的朋友阅读，本书对处理器的一些介绍，将有助于嵌入式开发。

内容安排

全书共15章，分为三篇，第一篇是理论篇，包含第1、2章，介绍了指令集架构、Verilog HDL的相关知识。第二篇是基础篇，包含第3～11章，采用增量模型，实现了教学版OpenMIPS处理器。第三篇是进阶篇，包含第12～15章，实现了实践版OpenMIPS处理器，并为其移植了嵌入式实时操作系统μC/OS-II。每章的主要内容如下。

- 第一篇 理论篇

第1章给出了计算机的简单组成模型、简单使用模型，对比了RISC与CISC，说明了指令集架构的作用，并列举了目前几种主要的指令集架构，由于OpenMIPS采用的是MIPS32指令集架构，所以本章后半部分重点介绍了MIPS32指令集架构。

第2章介绍了FPGA、Verilog HDL的基础知识，FPGA是可编程逻辑器件的一种，本书的实践版OpenMIPS处理器就将在FPGA上运行。

● 第二篇 基础篇

第3章介绍了教学版OpenMIPS处理器的设计蓝图，包括设计目标、处理器接口，以及最终完成时组成OpenMIPS的各个模块的作用，力图使读者有一个整体认识。另外，本章还详述了OpenMIPS处理器的实现方法。

第4章实现了OpenMIPS处理器的第一条指令ori，之所以选择这条指令作为我们实现的第一条指令，就是因为它足够简单，指令ori用来实现逻辑“或”运算，通过这条简单指令的实现，初步建立了OpenMIPS的五级流水线结构，后续章节实现其余指令的时候，都是在这个初步建立的流水线结构基础上进行扩充的。本章还建立了用于测试的小型SOPC，并通过ModelSim仿真验证ori指令、五级流水线实现的正确与否。

第5章讨论并解决了流水线数据的相关问题，然后修改第4章的OpenMIPS，添加实现了MIPS32指令集架构中定义的逻辑、移位操作与空指令。

第6章介绍添加实现了MIPS32指令集架构中定义的移动操作指令。

第7章介绍添加实现了MIPS32指令集架构中定义的算术操作指令。

第8章介绍添加实现了MIPS32指令集架构中定义的转移指令，OpenMIPS支持延迟转移。

第9章介绍添加实现了MIPS32指令集架构中定义的加载存储指令。

第10章介绍添加实现了MIPS32指令集架构中定义的协处理器CP0，以及协处理器访问指令。

第11章介绍添加实现了MIPS32指令集架构中定义的异常相关指令，并实现了异常处理。

在每一类指令的实现过程中，都是先介绍该类指令的格式、作用和用法，然后介绍实现思路，接着通过修改代码实现该类指令，最后，编写测试程序，使用仿真的方式验证实现的正确性。

• 第三篇 进阶篇

第12章介绍在教学版OpenMIPS处理器的基础上，通过添加Wishbone总线接口模块，实现了实践版OpenMIPS处理器。

第13章讲述设计实现了基于实践版OpenMIPS处理器的小型可编程片上系统SOPC的整个过程。该SOPC包括GPIO、UART控制器、Flash控制器、SDRAM控制器等模块，这些模块都具有Wishbone总线接口，与OpenMIPS处理器一起挂接在Wishbone总线互联矩阵上。

第14章将第13章实现的小型SOPC下载到实际的硬件平台上，编写测试程序，验证实践版OpenMIPS处理器实现的正确性。

第15章介绍了嵌入式实时操作系统μC/OS-II，并将其移植到本书设计的OpenMIPS处理器上，进一步验证了实践版OpenMIPS处理器实现的正确性，也为OpenMIPS处理器发挥实际作用奠定了基础。

本书特色

- **从无到有、从小到大，介绍一款处理器的成长过程**

在本书之前已有介绍软核处理器实现的书籍，这些书在介绍实现方法时有一个共同点：一次考虑所有的指令、所有的情况，然后给出代码。笔者认为这不是读者易于接受的一种方法，而且这也可能不是作者实现处理器时采用的方法。在本书中，笔者借鉴了软件开发中的“增量模型”的概念，使用了一种完全不同的实现方法：先考虑最简单的情况，给出代码，然后考虑稍微多一点的情况，修改、补充代码，随着考虑情况的增多，不停地修改、补充代码，最终，实现需求的两个版本。

- **教学、实践兼顾**

OpenMIPS处理器分为教学版和实践版两个版本。

教学版的主要设想是尽量简单，比如：在一个时钟周期内可以取到指令，完成存储、加载数据，这样处理器的运行情况（比如：流水线的运行）就比较理想化，与教科书相似，代码也很清晰简单，便于

使用其进行教学、学术研究和讨论，也有助于读者理解教科书上讲授的计算机的原理、计算机体系结构等知识。

实践版的主要设计思想是使OpenMIPS成为一个实际可用的处理器，能够下载到可编程逻辑器件上，运行实际有用的程序。为此，添加了Wishbone总线接口，这样就能方便地利用各种已有的SDRAM、Flash、GPIO、UART、LCD等模块控制器，组成一个SOPC，完成特定功能，进一步还可为其移植操作系统。

光盘内容

本书附带光盘提供了OpenMIPS的所有源代码，以及一些开发工具，详情如下。

- Code文件夹

提供了本书每一章涉及的OpenMIPS源代码、测试程序。

- Tools文件夹

提供了GNU工具链的安装文件，以及一个小工具Bin2Mem.exe，该工具用来将二进制数文件转化为可以用于ModelSim仿真的格式。

- Doc文件夹

提供了本书使用的一些IP核的说明手册，包括UART控制器、SDRAM控制器、GPIO模块等。还提供了FPGA开发平台DE2的说明手册。

- DE2文件夹

提供了用来将程序写入DE2上Flash芯片的工具，在第14、15章会用到。

致谢

感谢OC8051、OR1200、LEON3的开发者，正是你们的辛苦工作、无私奉献，使得我们有机会领略、学习这些优秀的作品，向你们致敬！

笔者是第二次与博文视点合作，一如既往的敏捷、迅速、干练，在此特别感谢孙学瑛老师，孙老师以专业的眼光审阅了全书，提出了许多宝贵意见，为本书的顺利出版耗费了不少心力。

感谢我的好友张世伟老师，为我提供了DE2开发平台。

最后，感谢我的爸爸、妈妈、姐姐、姐夫，以及可爱的外甥女，任何成绩的取得都离不开家人的身影，谢谢你们！

笔者学识有限，尽管对本书已通读数次，但仍不能保证书中无一纰漏，欢迎读者朋友对本书提出批评、建议，可以通过邮箱leishangwen@163.com与笔者交流。

写作体会

在实现OpenMIPS处理器的过程中，笔者深刻体会到“罗马非一日建成”这句话，外表看起来巨大、庞杂的罗马，也是通过人们一步一步、一天一天、一点一点建成的，处理器也是如此，读者首先不要被处理器的神秘吓到，从最简单的地方入手，逐步增加功能、完善设计，一行代码一行代码地书写，不仅要有实现处理器的远大目标，还要确立切实可操作的短期目标，比如本周实现除法指令，下周实现转移指令，诸如此类，等有一天你突然回头，会发现，原来已经走了那么远，实现了那么多功能。李白有诗云：两岸猿声啼不住，轻舟已过万重山。当是此意。

处理器实现了，但要把它实现过程明白地表达出来，让读者理解，则又是一件难事。笔者从开始写作到最终完稿的这一过程中，一直承受着巨大的煎熬，几度欲放弃写作，所幸的是最终坚持了下来。

最后，我想对各位读者说：

一个人的旅行是孤单的
一个人的冬季是寒冷的
但是
一个人的处理器是骄傲的
让我们骄傲一次

雷思磊

2014年8月

目 录

[前言](#)

[第一篇 理论篇](#)

[第1章 处理器与MIPS](#)

[1.1 计算机的简单模型](#)

[1.1.1 计算机的简单组成模型](#)

[1.1.2 计算机的简单使用模型](#)

[1.2 架构与指令集](#)

[1.2.1 CISC与RISC](#)

[1.2.2 主要的几种ISA](#)

[1.3 MIPS指令集架构的演变](#)

[1.4 MIPS32指令集架构简介](#)

[1.4.1 数据类型](#)

[1.4.2 寄存器](#)

[1.4.3 字节次序](#)

[1.4.4 指令格式](#)

[1.4.5 指令集](#)

[1.4.6 寻址方式](#)

[1.4.7 协处理器CP0](#)

[1.4.8 异常](#)

[1.5 本书的目标与组织方式](#)

[第2章 可编程逻辑器件与Verilog HDL](#)

[2.1 可编程逻辑器件概述](#)

[2.2 基于PLD的数字系统设计流程](#)

2.2.1 设计输入

2.2.2 综合

2.2.3 布局布线

2.2.4 下载

2.2.5 仿真

2.2.6 工具介绍

2.3 Verilog HDL简介

2.4 Verilog HDL中模块的结构

2.5 Verilog HDL基本要素

2.5.1 常量

2.5.2 变量声明与数据类型

2.5.3 向量

2.5.4 运算符

2.6 Verilog HDL行为语句

2.6.1 过程语句

2.6.2 赋值语句

2.6.3 条件语句

2.6.4 循环语句

2.6.5 编译指示语句

2.6.6 行为语句的可综合性

2.7 电路设计举例

2.8 仿真

2.8.1 系统函数

2.8.2 Test Bench

2.8.3 ModelSim仿真

2.9 本章小结

第二篇 基础篇

第3章 教学版OpenMIPS处理器蓝图

3.1 系统设计目标

3.1.1 设计目标

3.1.2 五级流水线

3.1.3 指令执行周期

3.2 教学版OpenMIPS处理器接口

3.3 文件说明

3.4 实现方法

第4章 第一条指令ori的实现

4.1 ori指令说明

4.2 流水线结构的建立

4.2.1 流水线的简单模型

4.2.2 原始的OpenMIPS五级流水线结构

4.2.3 一些宏定义

4.2.4 取指阶段的实现

4.2.5 译码阶段的实现

4.2.6 执行阶段的实现

4.2.7 访存阶段的实现

4.2.8 回写阶段的实现

4.2.9 顶层模块OpenMIPS的实现

4.3 验证OpenMIPS实现效果

4.3.1 指令存储器ROM的实现

4.3.2 最小SOPC的实现

4.3.3 编写测试程序

4.3.4 建立Test Bench文件

4.3.5 使用ModelSim检验OpenMIPS实现效果

4.4 MIPS编译环境的建立

4.4.1 VisualBox的安装与设置

4.4.2 GNU工具链的安装

4.4.3 使用GNU工具进行编译

4.4.4 使用GNU工具进行链接

4.4.5 得到ROM初始化文件

4.4.6 编写Makefile文件

4.5 第一条指令实现小结

第5章 逻辑、移位操作与空指令的实现

5.1 流水线数据相关问题

5.2 OpenMIPS对数据相关问题的解决措施

5.3 测试数据相关问题的解决效果

5.4 逻辑、移位操作与空指令说明

5.5 修改OpenMIPS以实现逻辑、移位操作与空指令

5.5.1 修改译码阶段的ID模块

5.5.2 修改执行阶段的EX模块

5.6 测试程序1——测试逻辑操作实现效果

5.7 测试程序2——测试移位操作与空指令实现效果

5.8 小结

第6章 移动操作指令的实现

6.1 移动操作指令说明

6.2 移动操作指令实现思路

6.2.1 新的数据相关情况的解决

6.2.2 系统结构的修改

6.3 修改OpenMIPS以实现移动操作指令

6.3.1 HI、LO寄存器的实现

6.3.2 修改译码阶段的ID模块

6.3.3 修改执行阶段

6.3.4 修改访存阶段

6.3.5 修改回写阶段

6.3.6 修改OpenMIPS顶层模块

6.4 测试程序

第7章 算术操作指令的实现

7.1 简单算术操作指令说明

7.2 简单算术操作指令实现思路

7.3 修改OpenMIPS以实现简单算术操作指令

7.3.1 修改译码阶段的ID模块

7.3.2 修改执行阶段的EX模块

7.4 测试简单算术操作指令实现效果

7.5 流水线暂停机制的设计与实现

7.5.1 流水线暂停机制的设计

7.5.2 流水线暂停机制的实现

7.6 乘累加、乘累减指令说明

7.7 乘累加、乘累减指令实现思路

7.8 修改OpenMIPS以实现乘累加、乘累减指令

7.8.1 修改译码阶段的ID模块

7.8.2 修改执行阶段的EX模块

7.8.3 修改EX/MEM模块

7.8.4 修改OpenMIPS模块

7.9 测试乘累加、乘累减指令实现效果

7.10 除法指令说明

7.11 除法指令实现思路

7.11.1 试商法

7.11.2 实现思路

7.11.3 系统结构的修改

7.12 修改OpenMIPS以实现除法指令

7.12.1 增加DIV模块

7.12.2 修改译码阶段的ID模块

7.12.3 修改执行阶段的EX模块

7.12.4 修改OpenMIPS模块

7.13 测试除法指令实现效果

7.14 数据流图的修改

第8章 转移指令的实现

8.1 延迟槽

8.2 转移指令说明

8.3 转移指令实现思路

8.3.1 实现思路

8.3.2 数据流图的修改

8.3.3 系统结构的修改

8.4 修改OpenMIPS以实现转移指令

8.4.1 修改取指阶段的PC模块

8.4.2 修改译码阶段

8.4.3 修改执行阶段的EX模块

8.4.4 修改OpenMIPS模块

8.5 测试转移指令的实现效果

8.5.1 测试跳转指令

8.5.2 测试分支指令

第9章 加载存储指令的实现

9.1 加载存储指令说明

9.1.1 加载指令lb、lbu、lh、lhu、lw说明

9.1.2 存储指令sb、sh、sw说明

9.1.3 加载存储指令用法示例

9.1.4 加载指令lw、lw说明

9.1.5 存储指令sw、sw说明

9.2 加载存储指令实现思路

9.2.1 数据流图的修改

9.2.2 系统结构的修改

9.3 修改OpenMIPS以实现加载存储指令

9.3.1 修改译码阶段

9.3.2 修改执行阶段

9.3.3 修改访存阶段

9.3.4 修改OpenMIPS顶层模块

9.4 修改最小SOPC

9.4.1 添加数据存储器RAM

9.4.2 修改最小SOPC

9.5 测试程序

9.6 链接加载指令ll、条件存储指令sc说明

9.7 ll、sc指令实现思路

9.7.1 ll、sc指令的实现

9.7.2 数据流图的修改

9.7.3 系统结构的修改

9.8 修改OpenMIPS以实现ll、sc指令

9.8.1 LLbit寄存器的实现

9.8.2 修改译码阶段的ID模块

9.8.3 修改访存阶段

9.8.4 修改OpenMIPS模块

9.9 测试ll、sc指令实现效果

9.10 load相关问题

9.10.1 load相关问题介绍

9.10.2 解决方法

9.11 修改OpenMIPS以解决load相关问题

9.11.1 修改译码阶段的ID模块

9.11.2 修改OpenMIPS模块

9.12 测试load相关问题解决效果

9.13 小结

第10章 协处理器访问指令的实现

10.1 协处理器介绍

10.2 协处理器CP0中的寄存器

10.3 协处理器CP0的实现

10.4 协处理器访问指令说明

10.5 协处理器访问指令实现思路

10.5.1 实现思路

10.5.2 数据流图的修改

10.5.3 系统结构的修改

10.6 修改OpenMIPS以实现协处理器访问指令

10.6.1 修改译码阶段

10.6.2 修改执行阶段

10.6.3 修改访存阶段

10.6.4 修改OpenMIPS模块

10.7 测试程序

第11章 异常相关指令的实现

11.1 MIPS32架构中定义的异常类型

11.2 精确异常

11.3 异常处理过程

11.4 异常相关指令介绍

11.4.1 自陷指令

11.4.2 系统调用指令syscall

11.4.3 异常返回指令eret

11.5 异常处理实现思路

11.5.1 实现思路

11.5.2 修改数据流图

11.5.3 修改系统结构

11.6 修改OpenMIPS以实现异常处理

11.6.1 修改取指阶段

11.6.2 修改译码阶段

11.6.3 修改执行阶段

11.6.4 修改访存阶段

11.6.5 修改协处理器CP0

11.6.6 修改控制模块CTRL

11.6.7 修改OpenMIPS

11.7 再次修改最小SOPC

11.8 测试程序

11.8.1 测试程序1——测试系统调用异常

11.8.2 测试程序2——测试自陷异常

11.8.3 测试程序3——测试时钟中断

11.9 教学版OpenMIPS处理器实现小结

第三篇 进阶篇

第12章 实践版OpenMIPS处理器设计与实现

12.1 实践版OpenMIPS处理器的设计目标

12.2 Wishbone总线介绍

12.2.1 Wishbone总线接口说明

12.2.2 Wishbone总线单次读操作的过程

12.2.3 Wishbone总线单次写操作的过程

- 12.2.4 SEL O/SEL I信号说明
 - 12.3 实践版OpenMIPS处理器接口
 - 12.4 实践版OpenMIPS处理器的实现思路
 - 12.5 从教学版OpenMIPS到实践版OpenMIPS
 - 12.5.1 Wishbone总线接口模块的实现
 - 12.5.2 修改CTRL模块
 - 12.5.3 修改OpenMIPS顶层模块
 - 12.6 实践版OpenMIPS处理器实现小结
- 第13章 基于实践版OpenMIPS的小型SOPC
- 13.1 小型SOPC的结构
 - 13.2 Wishbone总线互联矩阵WB CONMAX
 - 13.3 GPIO
 - 13.4 UART控制器
 - 13.4.1 UART简介
 - 13.4.2 UART16550 IP核介绍
 - 13.5 Flash控制器
 - 13.5.1 Flash简介
 - 13.5.2 Flash控制器的设计
 - 13.5.3 Flash控制器的实现
 - 13.6 SDRAM控制器
 - 13.6.1 SDRAM简介
 - 13.6.2 SDRAM CONTROLLER IP核
 - 13.7 实现基于实践版OpenMIPS的小型SOPC
 - 13.8 本章小结
- 第14章 验证实践版OpenMIPS处理器
- 14.1 DE2平台简介
 - 14.2 测试需要的硬件连接

14.3 QuartusII工程建立

14.4 测试步骤说明

14.5 测试——GPIO实验

14.5.1 测试内容

14.5.2 测试程序

14.5.3 编译测试程序

14.5.4 将测试程序写入Flash芯片

14.5.5 下载小型SOPC到DE2

14.5.6 测试效果

14.6 测试二——UART实验

14.6.1 测试内容

14.6.2 测试程序

14.6.3 测试效果

14.7 测试三——模拟操作系统的加载过程

14.7.1 测试内容

14.7.2 测试程序BootLoader

14.7.3 测试程序SimpleOS

14.7.4 将测试程序写入Flash

14.7.5 测试效果

14.8 本章小结

第15章 为OpenMIPS处理器移植μC/OS-II

15.1 为什么需要操作系统

15.2 嵌入式实时操作系统介绍

15.3 μC/OS-II简介

15.4 μC/OS-II特点

15.5 μC/OS-II的几个概念

15.5.1 任务

15.5.2 任务调度

15.5.3 任务切换

15.5.4 μC/OS-II的中断处理

15.5.5 时钟节拍

15.5.6 μC/OS-II的初始化

15.5.7 μC/OS-II的启动

15.6 μC/OS-II的基本功能

15.6.1 任务间的通信与同步

15.6.2 任务管理

15.6.3 时间管理

15.6.4 内存管理

15.7 μC/OS-II的文件体系

15.8 μC/OS-II的移植条件

15.9 C语言中使用汇编代码

15.10 MIPS函数调用规范

15.10.1 寄存器使用规范

15.10.2 参数传递

15.10.3 函数返回值

15.10.4 堆栈布局

15.10.5 示例

15.11 μC/OS-II在OpenMIPS处理器上的移植

15.11.1 文件目录的建立

15.11.2 修改os_cpu.h文件

15.11.3 修改os_cpu_a.S文件

15.11.4 修改os_cpu_c.c文件

15.12 测试程序

15.12.1 创建openmips.h文件

15.12.2 创建openmips.c文件

15.13 编译指示文件的建立

15.14 OpenMIPS处理器运行移植后的μC/OS-II

15.15 本章小结

附录A 教学版OpenMIPS各个模块的接口说明

A.1 PC模块接口说明

A.2 IF/ID模块接口说明

A.3 ID模块接口说明

A.4 Regfile模块接口说明

A.5 ID/EX模块接口说明

A.6 EX模块接口说明

A.7 DIV模块接口说明

A.8 EX/MEM模块接口说明

A.9 MEM模块接口说明

A.10 MEM/WB模块接口说明

A.11 CP0模块接口说明

A.12 LLbit模块接口说明

A.13 HILO模块接口说明

A.14 CTRL模块接口说明

附录B OpenMIPS实现的所有指令及对应的机器码

B.1 逻辑操作指令

B.2 移位操作指令

B.3 移动操作指令

B.4 算术操作指令

B.5 转移指令

B.6 加载存储指令

B.7 协处理器访问指令

B.8 异常相关指令

B.9 空指令及其他指令

参考文献

第一篇 理论篇

第1章 处理器与MIPS

第2章 可编程逻辑器件与Verilog HDL

第1章 处理器与MIPS

时间开始了！

——胡风·1949

让我们以一句诗意的话，开始本书的阅读。

时间从1971年11月15日开始，那一天，Intel发布了世界上第一款单芯片微处理器4004。

1.1 计算机的简单模型

计算机很复杂，可以听歌、看电影、上网、玩游戏，内部是怎么工作的，这个问题太可怕了，太复杂了。

计算机很简单，只可以做加、减、乘、除、逻辑、移位、转移、存储、加载等几类的操作，太简单了。

复杂？简单？其实取决于个人对事物的认识程度，认识得越多，了解得越深刻，那么就越接近本质，而本质往往都是简单的，比如大名鼎鼎的质能方程，一个简单的式子就解释了质量与能量的关系。

计算机就是一台计算的设备，而且是一台很基础的计算设备，只能计算小学数学课堂上讲授的四则运算，再加上一些并不复杂的与、或、非等逻辑运算，其余诸如平方、开方、微分、积分等都是做不了的。有读者会有疑惑，你说得太简单了吧，别急，且听我慢慢道来。

1.1.1 计算机的简单组成模型

计算机的组成有三大部分：处理器（Central Processing Unit，CPU）、输入输出（Input/Output，I/O）、存储器（Memory）。处理器从存储器中获取指令，然后按照指令执行一定的操作，输入/输出用来提供运算数据、显示运算结果。如图1-1所示。

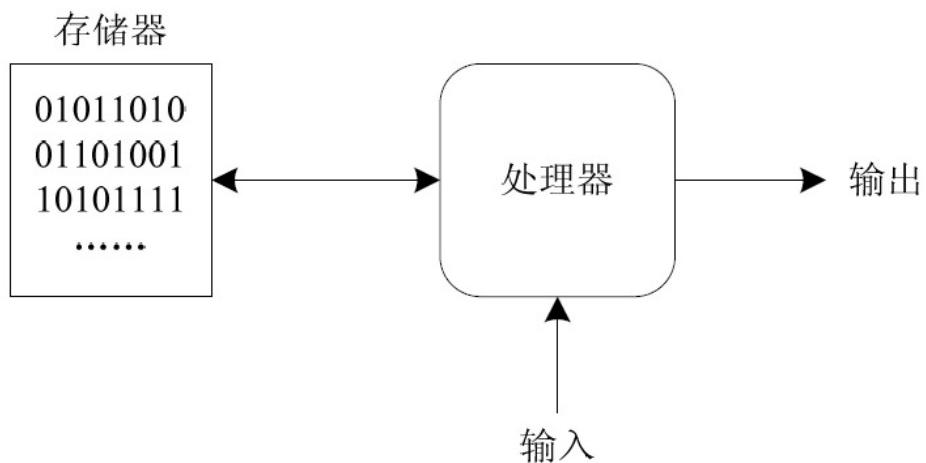


图1-1 计算机的简单组成模型

存储器中存储的是指令，指令就是一条运算命令，比如：将a与b相加，结果存储到c中，处理器按照命令执行即可。目前的计算机是一个二进制的世界，所有的信息都是用0、1组合来表示的，所以一条指令就是一串0、1编码，正如图1-1所示。处理器内部具有译码功能，用来解释接收到的0、1编码表示的运算类型，据此进行运算。

1.1.2 计算机的简单使用模型

我们使用计算机上网、办公，都是通过一定的应用程序实现的，而这些应用程序实际就是一批指令的集合，当然，这里的“一批”指的是指令的数目庞大，实际上种类是非常少的，只有几百条，常用的也就几十条。通过这些指令的组织、配合，就实现了目前丰富多彩的应用。

理论上，可以直接使用0、1编码进行程序设计，但是那样显然太不方便、容易出错，于是人们使用一些助记符来表示各种指令，这就是汇编指令，使用汇编程序将汇编指令翻译为计算机可以识别的0、1编码，后来，又发明了高级语言，其语法、使用方式比汇编更加方便、更加易于理解。一般使用编译程序将高级语言编写的程序翻译为汇编指令，然后再使用汇编程序将其翻译为0、1编码。本质上是一样的。如图1-2所示。

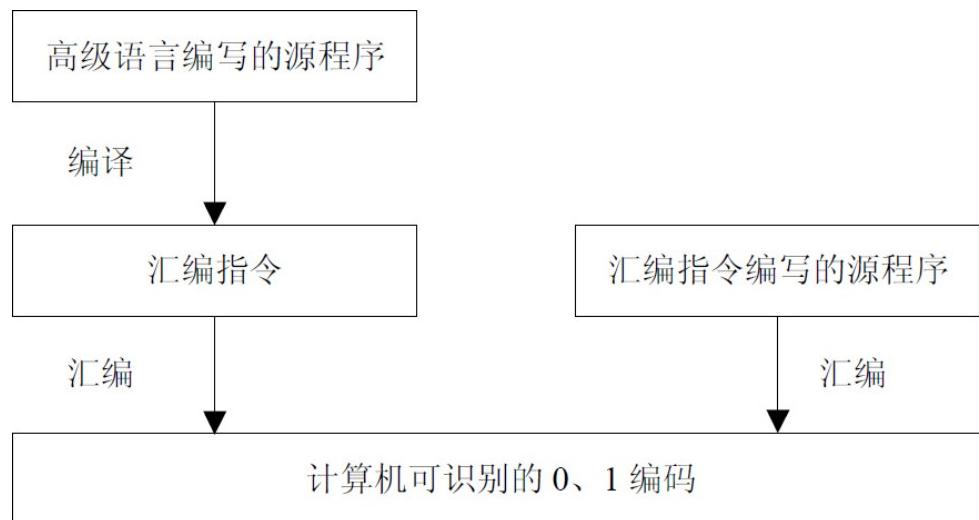


图1-2 计算机的简单使用模型

这就是计算机的简单使用模型，无论是视频软件、浏览器，还是其他任何软件；无论是使用C#开发、Java开发，还是使用任何其他语言开发；无论是在Windows环境下运行、Android环境下运行，还是在

其他任何平台下运行；无论是在ARM处理器上运行、Intel处理器上运行，还是在其他任何处理器上运行；无论是在银河二号这样的大型机运行、个人使用的PC上运行，还是在其他任何机器上运行，都遵循这样的一套使用模型。

总结一下，计算机只识别0、1编码串，任何程序最终都要转变为0、1编码串，而且0、1编码的种类是有限的，计算机按照0、1编码的命令进行工作。这样说，读者应该觉得计算机很简单了吧。

在计算机中处于核心地位的是处理器，也称为CPU，作用是识别0、1编码，据此进行各种运算和数据处理。**本书的目标就是实现一个CPU**。

1.2 架构与指令集

类似于不同国家的人使用不同的文字，不同的处理器也使用不同的指令，这样，为处理器A编写的程序不能直接在处理器B上使用，需要重新编写，然后再次编译、汇编后才可使用，减低了软件的移植性。显然，极为不便。

IBM为了让自己的一系列计算机能使用相同的软件，免去重复编写软件的痛苦，在它的System/360计算机中引入了指令集架构（Instruction Set Architecture，ISA）的概念，将编程所需要了解的硬件信息从硬件系统中抽象出来，这样软件人员就可以面向ISA进行编程，开发出来的软件不经过修改就可以应用在符合该ISA的所有计算机上。ISA用来描述编程时用到的抽象机器，而非这种机器的具体实

现，从软件人员的角度来看，ISA包括一套指令集和一些寄存器，知道它们就可以编写程序了。

与ISA对应的一个概念是微架构（Microarchitecture），后者是前者的一个实现，比如Intel的许多处理器都是遵循x86的ISA，但是每一款处理器都有自己的微架构。ISA好比是设计规范，微架构则是具体实现，同样的ISA，不同的微架构，会带来不同的性能。

1.2.1 CISC与RISC

从大的方面，根据ISA的不同可以将计算机分为两类：复杂指令集计算机（Complex Instruction Set Computer，CISC）和精简指令集计算机（Reduced Instruction Set Computer，RISC）。它们的主要区别是，CISC的每条指令对应的0、1编码串长度不一，而RISC的每条指令对应的0、1编码串长度是固定的。

在计算机发展的早期，人们使用汇编语言编程，偏好强大好用的指令集，处理器的设计人员于是将指令集设计得更强大、更灵活，并且那个时期的存储器既昂贵且速度慢，因此指令使用了变长编码，以节约存储空间，由于一条指令就能完成很多功能，从而减少了对内存的访问次数，这样也减少了缓慢的存储器访问对程序性能的影响。典型的CISC指令集架构就是Intel的x86 ISA。上世纪70年代中期，人们发现CISC指令集中的各种指令，其使用频率相差悬殊，大约有20%的指令会被反复使用，占整个程序代码的80%。而余下80%的指令却不经常使用，只占整个程序代码的20%，显然，这种结构是不太合理的。于是人们提出将指令集和处理器进行重新设计，减少那些使用不多的指令，只保留常用的简单指令，这样处理器就不需要浪费太多的晶体

管去做那些很复杂又很少使用的功能，于是产生了RISC。1979年美国加州大学伯克利分校的David Patterson首先提出了RISC的概念，RISC并不只是简单地减少指令，更主要的目的是研究如何使计算机的结构更加简单合理以提高运算速度。其特点是指令长度固定、指令格式种类少、寻址方式种类少、大量使用寄存器等。由于在RISC中使用的指令大多数是简单指令且都能在一个时钟周期内完成，因而处理器的频率得以大幅提升，同时易于设计流水线。RISC是计算机发展历史上的一个里程碑，以致有人开玩笑地把RISC定义为：1985年之后发布的所有处理器。

Intel也尝试做RISC处理器，但是因为兼容性问题，没有成功，后来在1995年，Intel的David B.Papworth和他的同事一起设计了Pentium Pro处理器，在这个处理器中，x86指令先被解码为类似于RISC指令的微操作（microoperation，简称为uops），之后的执行过程采用RISC内核，这种方式一直延续至今。

1.2.2 主要的几种ISA

目前并没有一种统一的ISA为各个处理器厂商所接受，而是存在多种ISA，就像这个世界存在多种语言一样，但是主要的语言只有几种：汉语、英语、法语、俄语等。主要的ISA也只有几种：x86、ARM、SPARC、POWER、MIPS，除了x86是CISC ISA外，其余都是RISC ISA。

1. x86

x86架构于1978年推出的Intel 8086处理器中首度出现，三年后，Intel 8086为IBM PC所选用，之后x86便成为了个人计算机的标准平台，成为了历史上最成功的指令集架构。目前绝大多数个人计算机使用的都是兼容x86指令集架构的处理器。

2. ARM

1985年，英国的Acorn公司设计了自己的第一代32位、6MHz处理器，命名为Acorn RISC Machine，简称为ARM1。1990年，由苹果公司、VLSI公司共同出资，改组Acorn为ARM计算机公司，同时不再涉足具体芯片生产，只出售IP核。ARM公司设计低功耗、高性能的CPU内核，然后授权给其他公司，后者设计生产具体的处理器芯片。

由于ARM侧重于低功耗、低成本，主要面向的是嵌入式应用，故随着智能手机、平板等移动设备的普及，ARM公司发展得非常迅速。

ARM架构从v4、v4T、v5、v5E、v6，发展到v7，其中v7又分为v7-A、v7-R、v7-M等多种，苹果公司的A9处理器采用的就是ARM v7-A架构。

3. SPARC

SPARC（Scalable Processor ARChitecture，可扩展处理器架构）源自美国加州大学伯克利分校上世纪80年代的研究，由Sun公司在1985年首先提出。1989年成为商用架构，生产出SPARC系列的处理器，Sun将其用在高性能工作站和服务器上。SPARC架构目前的版本有v8、v9。

SPARC架构对外完全开放，在此基础上出现了一些开放源代码的处理器，比如：Sun公司的UltraSPARC T1、LEON等。其中，LEON是一种SPARC v8架构的处理器，至今已发布到了LEON4。最初的LEON1与LEON2由欧洲航天局发布，LEON3由Gaisler Research公司设计发布，2008年Aeroflex收购了Gaisler Research公司，并于2010年1月发布了LEON4，不过LEON4至今还没有公布源代码。LEON系列使用VHDL编写代码，原计划是使用在航天器上。

4. POWER

POWER（Performance Optimization With Enhanced RISC）是由IBM公司设计开发的一种RISC结构的指令集架构。IBM生产的很多服务器、大型机、小型机及工作站都采用POWER架构的微处理器作为其主CPU使用。

1991年Apple、IBM和Motorola三家公司成立了AIM联盟（AIM为Apple、IBM、Motorola的首字母），对POWER架构进行了修改，形成了PowerPC架构。2004年IBM发起了Power.org联盟，发布了统一的指令集架构，将POWER与PowerPC统一到新的Power架构中。

5. MIPS

MIPS的含义是无内锁流水线微处理器（Microprocessor without Interlocked Piped Stages），是上世纪80年代诞生的RISC CPU的重要代表，其设计者John Hennessy时任斯坦福大学的教授。当初的设计基于以下理念：使用相对简单的指令，结合优秀的编译器以及采用流水线执行指令的硬件，就可以用更少的晶元面积生产更快的处理器。这一理念是如此的成功，以至于1984年就成立了MIPS计算机系统公司对

MIPS架构进行商业化。在随后的十几年中，MIPS架构在很多方面得到发展，在工作站和服务器系统中得到了很多应用。MIPS架构也从MIPS I、MIPS II、MIPS III、MIPS IV、MIPS V、MIPS32发展到MIPS64。John Hennessy与RISC概念的提出者David Patterson合著了计算机领域影响甚广的教科书《计算机体系结构——量化研究方法》，该书至今已出到第五版。

国内的龙芯处理器采用的就是MIPS架构。本书计划实现的处理器也采用MIPS架构，这里涉及两个问题。

(1) 为什么要采用一个现有的指令集架构？

这是因为现有的指令集架构已经形成了一套完善的环境，其中既有成熟的编译器，还有大量的应用程序，采用现有的指令集架构，都可以直接使用这些环境。反之，如果设计自己独有的一套指令集架构，那么编译器、应用软件都需要自己重新开发，工作量巨大。还是以语言作比喻，一个人当然可以发明、使用自己独有的语言，但是如何与别人交流呢？无法与人交流，再优秀的语言也注定会消失。

(2) 为什么要采用MIPS架构？

首先MIPS的设计是RISC架构中的经典之作，很多处理器都吸收了其中的设计思想；其次，MIPS架构中指令的专利期已过，可以自由使用。

本章接下来将重点介绍MIPS指令集架构。

1.3 MIPS指令集架构的演变

MIPS指令集架构自上世纪80年代出现后，一直在进行更新换代，从最初的MIPS I到MIPS V，发展到可支持扩展模块的MIPS32、MIPS64系列，再到集成代码压缩技术的microMIPS32、microMIPS64。每个MIPS ISA都是其前一个的超集，没有任何遗漏，只有增加新的功能。

1. MIPS I

提供加载/存储、计算、跳转、分支、协处理及其他特殊指令。该指令集架构用于最初的MIPS处理器R2000/R3000。R2000是1985年推出的首款MIPS CPU，由110000个晶体管组成，是一个8MHz的32位处理器。R3000是R2000的下一代产品，与前者相比仅仅是时钟频率不同而已。

2. MIPS II

增加了自陷指令、链接加载指令、条件存储指令、同步指令、可能分支指令、平方根指令。最初计划用在MIPS处理器R6000上，但由于工艺选择的问题，R6000从1988年开始设计后，就一直问题不断，最终未能大规模生产。但MIPS II指令集架构是后期MIPS32指令集架构的直接先驱。

3. MIPS III

提供了32位指令集，同时支持64位指令集。最初用于MIPS处理器R4000。R4000是于1991年推出的64位处理器，首次加入了浮点处理器单元，主时钟频率提高到了100MHz。后来出现了一系列的R4000处理器。

4. MIPS IV

在MIPS III基础上增加了条件移动指令、预取指令以及一些浮点指令。最初用于MIPS处理器R8000，后来应用于R5000/R10000。R5000与R10000虽然使用相同的指令集架构，但是两者微架构的设计理念完全不同。R5000于1995年推出，采用的是经典的五级流水线、顺序执行。R10000于1996年推出，采用的是乱序执行。

5. MIPS V

在MIPS IV的基础上增加了可以提高代码生产效率和数据转移效率的指令。但是没有任何一个处理器基于该架构。MIPS V指令集架构是后期MIPS64指令集架构的直接先驱。

6. MIPS32/64

MIPS32/64于1998年提出，MIPS32以MIPS II架构为基础，选择性地加入了MIPS III、MIPS IV、MIPS V，提高了代码生成和数据移动的效率。MIPS64以MIPS V架构为基础，同时兼容MIPS32。该架构第一次包含了被称为协处理器0的“CPU控制”功能。1999年以后设计的大多数MIPS处理器都与该标准兼容。2003年，发布了MIPS32/64指令集架构的第二版（Release 2），也称为MIPS32/64 R2。最新的是第五版（Release 5），也称为MIPS32/64 R5。但目前广泛使用的是第二版，非常成功的MIPS 4K、24K系列处理器遵循的就是MIPS32 R2架构。

MIPS32/64在基本指令的基础上，还提供了一些面向特定应用的指令，这些指令采用特定应用扩展（Application-Specific Extensions，ASE）的形式。一种处理器是否实现了某种扩展，可以通过设置标准的配置寄存器指明。主要的扩展列举如下。

- MIPS 16e：是专门为嵌入式系统及存储空间有限情况下的应用而设计的，可以在一个程序中执行16位和32位两种混合长度的指令，能使最终代码长度减少40%。MIPS32、MIPS64都支持MIPS 16e。
- SmartMIPS：是为了满足智能卡和灵活小系统的市场需要而设计的，是一套能高效节省存储空间的扩展指令集，此外还能提高智能卡领域非常关键的加密运算的性能。MIPS32支持SmartMIPS。
- MIPS-3D：提供了更好的几何运算处理，具有成对单精度数据类型，还提供专用指令来加快对该类型数据的处理。MIPS64支持MIPS-3D，MIPS32第二版也支持MIPS-3D。
- MCU：Micro-Control Unit微控制单元，增强了内存映射I/O的处理、提供了更低的中断延迟。MIPS32、MIPS64都支持MCU。

7. microMIPS32/64

microMIPS32/64指令集架构集成了16位和32位优化指令的高性能代码压缩技术，保持了98%的MIPS32性能，同时至少减少了30%的代码体积，从而降低芯片成本，也有助于降低系统功耗。MIPS M14K内核是MIPS科技于2009年发布的首款遵循microMIPS指令集架构的MIPS32兼容内核。

MIPS指令集架构的演变可以使用图1-3描述。注意图中没有Release 4，这是因为对于很多人来说，4是个不吉利的数字，所以MIPS没有发布Release 4，而是直接发布Release 5。

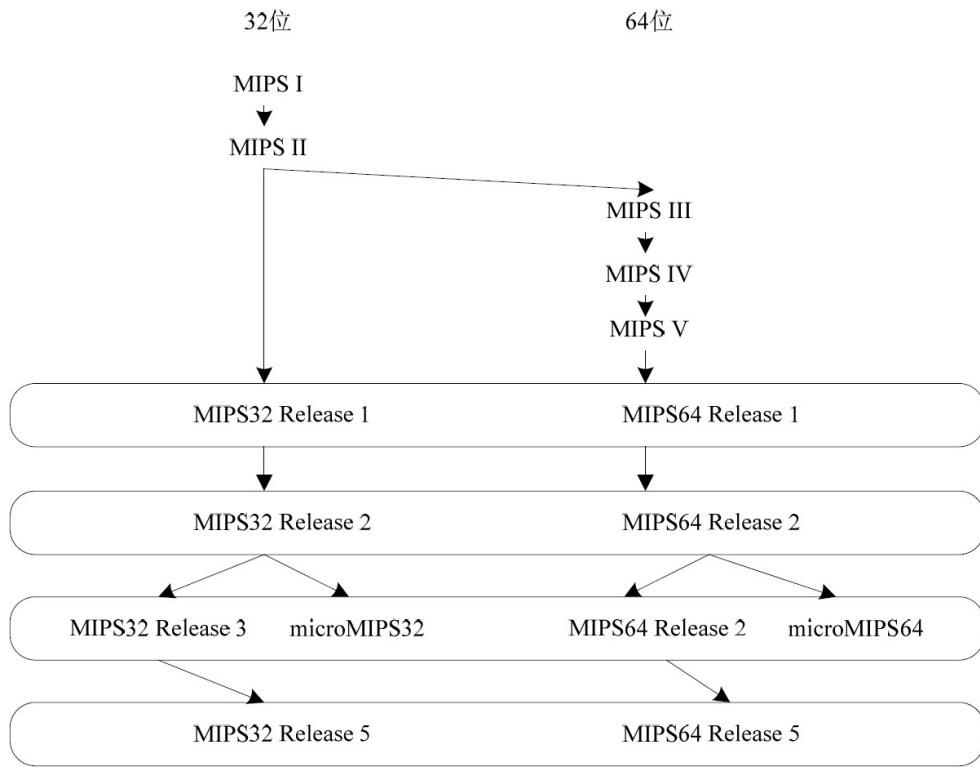


图1-3 MIPS指令集架构的演变

1.4 MIPS32指令集架构简介

本书设计的处理器遵循MIPS32 Release 1架构，所以本节介绍的MIPS32指令集架构指的就是MIPS32 Release 1。

1.4.1 数据类型

指令的主要任务就是对操作数进行运算，操作数有不同的类型和长度，MIPS32提供的基本数据类型如下。

- 位 (b)：长度是1bit。
- 字节 (Byte)：长度是8bit。

- 半字 (Half Word) : 长度是16bit。
- 字 (Word) : 长度是32bit。
- 双字 (Double Word) : 长度是64bit。

此外，还有32位单精度浮点数、64位双精度浮点数等。

1.4.2 寄存器

在前文介绍RISC的特点时提到一点：大量使用寄存器。这是因为寄存器的存取可以在一个时钟周期内完成，同时也简化了寻址方式。MIPS32的指令中除加载/存储指令外，都是使用寄存器或立即数作为操作数的。MIPS32中的寄存器分为两类：通用寄存器（General Purpose Register, GPR）、特殊寄存器。

1. 通用寄存器

MIPS32架构定义了32个通用寄存器，使用\$0、\$1...\$31表示，都是32位。其中\$0一般用做常量0。

在硬件上没有强制指定寄存器的使用规则，但是在实际使用中，这些寄存器的用法都遵循一系列约定，例如：寄存器\$31一般存放子程序的返回地址。MIPS32中通用寄存器的约定用法如表1-1所示。在本书大部分章节中，测试程序都是直接使用汇编指令编写的，对寄存器的约定用法还不需要十分在意，但是本书的最后一章移植μC/OS-II时，因为涉及C语言、汇编混合编程，对寄存器的约定用法就需要十分在意了。读者届时可以体会表1-1中各个寄存器的约定用法。

表1-1 MIPS32中通用寄存器的约定用法

寄存器名字	约定命名	用途
\$0	zero	总是为0
\$1	at	留作汇编器生成一些合成指令
\$2、\$3	v0、v1	用来存放子程序返回值
\$4~\$7	a0~a3	调用子程序时，使用这4个寄存器传输前4个非浮点参数
\$8~\$15	t0~t7	临时寄存器，子程序使用时可以不用存储和恢复
\$16~\$23	s0~s7	子程序寄存器变量，改变这些寄存器值的子程序必须存储旧的值并在退出前恢复，对调用程序来说值不变
\$24、\$25	t8、t9	临时寄存器，子程序使用时可以不用存储和恢复
\$26、\$27	\$k0、\$k1	由异常处理程序使用
\$28或\$gp	gp	全局指针
\$29或\$sp	sp	堆栈指针
\$30或\$fp	s8/fp	子程序可以用来作堆栈帧指针
\$31	ra	存放子程序返回地址

2. 特殊寄存器

MIPS32架构中定义的特殊寄存器有三个：PC（Program Counter程序计数器）、HI（乘除结果高位寄存器）、LO（乘除结果低位寄存器）。进行乘法运算时，HI和LO保存乘法运算的结果，其中HI存储高32位，LO存储低32位；进行除法运算时，HI和LO保存除法运算的结果，其中HI存储余数，LO存储商。

1.4.3 字节次序

数据在存储器中是按照字节存放的，处理器也是按照字节访问存储器中的指令或数据，但是如果需要读出一个字，也就是4个字节，比如读出的是 $\text{mem}[n]$ 、 $\text{mem}[n+1]$ 、 $\text{mem}[n+2]$ 、 $\text{mem}[n+3]$ 这四个字节，那么最终交给处理器的有两种结果。

- $\{\text{mem}[n], \text{mem}[n+1], \text{mem}[n+2], \text{mem}[n+3]\}$
- $\{\text{mem}[n+3], \text{mem}[n+2], \text{mem}[n+1], \text{mem}[n]\}$

前者称为大端模式（Big-Endian），也称为MSB（Most Significant Byte），后者称为小端模式（Little-Endian），也称为LSB（Least Significant Byte）。在大端模式下，数据的高位保存在存储器的低地址中，而数据的低位保存在存储器的高地址中。图1-4给出0x12345678在两种模式下的存储情况。本书实现的处理器采用的是大端模式（Big-Endian）。

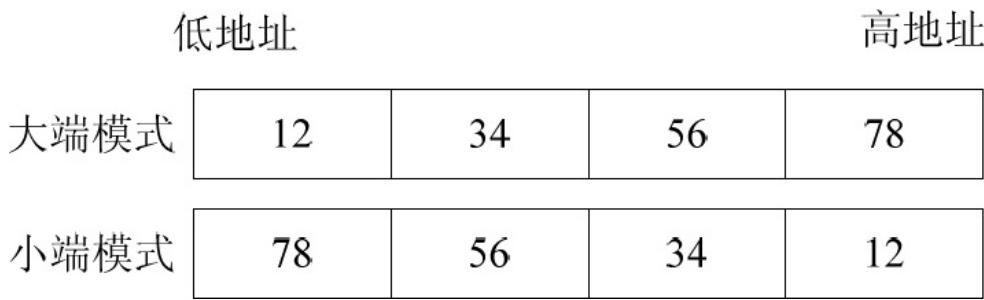


图1-4 大、小端模式下存储0x12345678的区别

1.4.4 指令格式

MIPS32架构中的所有指令都是32位，也就是32个0、1编码连在一起表示一条指令，有三种指令格式。如图1-5所示。其中op是指令码、func是功能码。

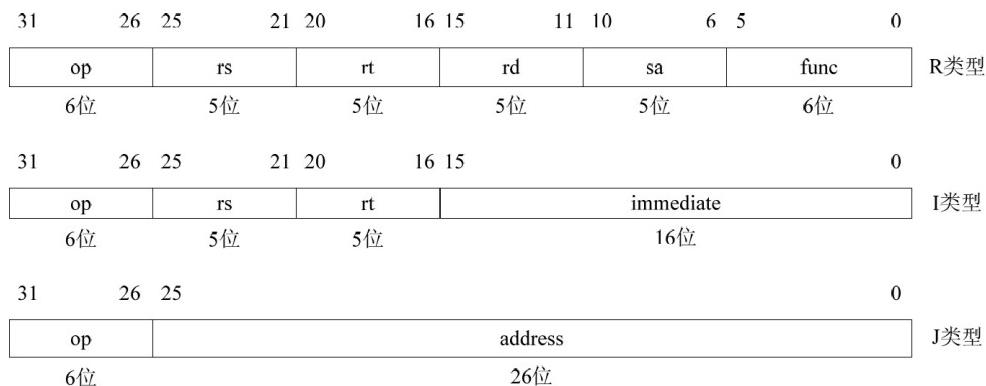


图1-5 MIPS32架构中的三种指令格式

(1) R类型：具体操作由op、func结合指定，rs和rt是源寄存器的编号，rd是目的寄存器的编号，比如：假设目的寄存器是\$3，那么对应的rd就是00011（此处是二进制数）。MIPS32架构中有32个通用寄存器，使用5位编码就可以全部表示，所以rs、rt、rd的宽度都是5位。sa只有在移位指令中使用，用来指定移位位数。

(2) I类型：具体操作由op指定，指令的低16位是立即数，运算时要将其扩展至32位，然后作为其中一个源操作数参与运算。

(3) J类型：具体操作由op指定，一般是跳转指令，低26位是字地址，用于产生跳转的目标地址。

1.4.5 指令集

在“计算机的简单使用模型”中已经介绍过，可以直接使用0、1编码进行程序设计，但是那样显然太不方便、容易出错，于是人们使用一些助记符来表示各种指令，这就是汇编指令，使用汇编程序将汇编指令翻译为计算机可以识别的0、1编码。也就是将汇编指令翻译为图1-5所示的格式，这样处理器就可以识别了。MIPS32架构中定义的指令可以分为以下几类。注意：其中不包括浮点指令，因为本书实现的处理器不包含浮点处理单元，也就没有实现浮点指令，所以此处不介绍浮点指令。

1. 逻辑操作指令

有8条指令：and、andi、or、ori、xor、xori、nor、lui，实现逻辑与、或、异或、或非等运算。本书设计的处理器实现了所有逻辑操作指令，将在第4、5章详细介绍各个逻辑操作指令的格式、作用、用法，以及实现过程。

2. 移位操作指令

有6条指令：sll、sllv、sra、srav、srl、sriv。实现逻辑左移、右移、算术右移等运算。本书设计的处理器实现了所有移位操作指令，

将在第5章详细介绍各个移位操作指令的格式、作用、用法，以及实现过程。

3. 移动操作指令

有6条指令：movn、movz、mfhi、mthi、mflo、mtlo，用于通用寄存器之间的数据移动，以及通用寄存器与HI、LO寄存器之间的数据移动。本书设计的处理器实现了所有移动操作指令，将在第6章详细介绍各个移动操作指令的格式、作用、用法，以及实现过程。

4. 算术操作指令

有21条指令：add、addi、addiu、addu、sub、subu、clo、clz、slt、slti、sltiu、sltu、mul、multu、madd、maddu、msub、msubu、div、divu，实现了加法、减法、比较、乘法、乘累加、除法等运算。本书设计的处理器实现了所有算术操作指令，将在第7章详细介绍各个算术操作指令的格式、作用、用法，以及实现过程。

5. 转移指令

有14条指令：jr、jalr、j、jal、b、bal、beq、bgez、bgezal、bgtz、blez、bltz、bltzal、bne，其中既有无条件转移，也有条件转移，用于程序转移到另一个地方执行。本书设计的处理器实现了所有转移指令，将在第8章详细介绍各个转移指令的格式、作用、用法，以及实现过程。

6. 加载存储指令

有14条指令：lb、lbu、lh、lhu、ll、lw、lwl、lwr、sb、sc、sh、sw、swl、swr，以“l”开始的都是加载指令，以“s”开始的都是存储指

令，这些指令用于从存储器中读取数据，或者向存储器中保存数据。本书设计的处理器实现了所有加载存储指令，将在第9章详细介绍各个加载存储指令的格式、作用、用法，以及实现过程。

7. 协处理器访问指令

有2条指令：mtc0、mfc0，用于读取协处理器CP0中某个寄存器的值，或者将数据保存到协处理器CP0中的某个寄存器。本书设计的处理器实现了所有协处理器访问指令，将在第10章详细介绍协处理器、协处理器访问指令的格式、作用、用法，以及实现过程。

8. 异常相关指令

有14条指令，其中有12条自陷指令，包括：teq、tge、tgeu、slt、sltlu、tne、teqi、tgei、tgeiu、lti、ltiul、tnei，此外还有系统调用指令syscall、异常返回指令eret。本书设计的处理器实现了所有异常相关指令，将在第11章详细介绍异常相关指令的格式、作用、用法，以及实现过程。

9. 其余指令

有4条指令：nop、ssnop、sync、pref，其中nop是空指令，ssnop是一种特殊类型的空指令，sync指令用于保证加载、存储操作的顺序，pref指令用于缓存预取。本书设计的处理器对这4条指令进行了简化并加以实现，将在第5章详细介绍简化后的实现过程。

1.4.6 寻址方式

MIPS32架构的寻址模式有寄存器寻址、立即数寻址、寄存器相对寻址和PC相对寻址四种。其中寄存器相对寻址、PC相对寻址的介绍如下。

(1) 寄存器相对寻址

这种寻址模式主要是加载/存储指令使用，其将一个16位的立即数做符号扩展，然后与指定通用寄存器的值相加，从而得到一个有效地址，如图1-6所示。

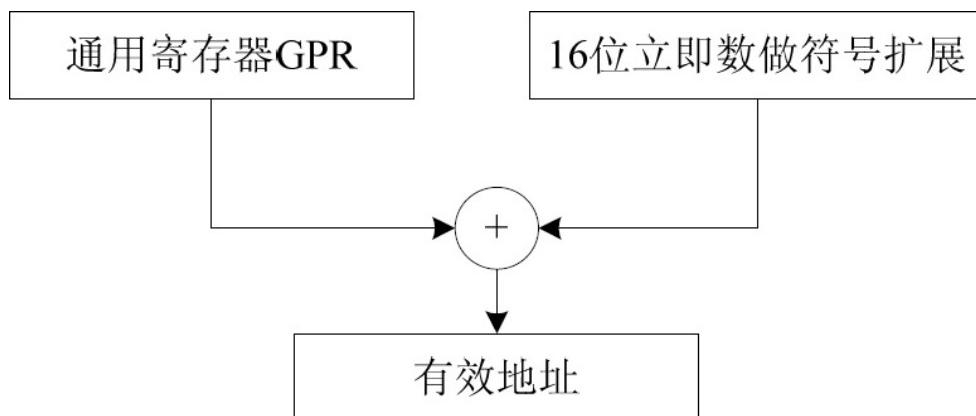


图1-6 寄存器相对寻址

(2) PC相对寻址

这种寻址模式主要是转移指令使用，在转移指令中有一个16位的立即数，将其左移两位并作符号扩展，然后与程序计数寄存器PC的值相加，从而获得有效地址。如图1-7所示。

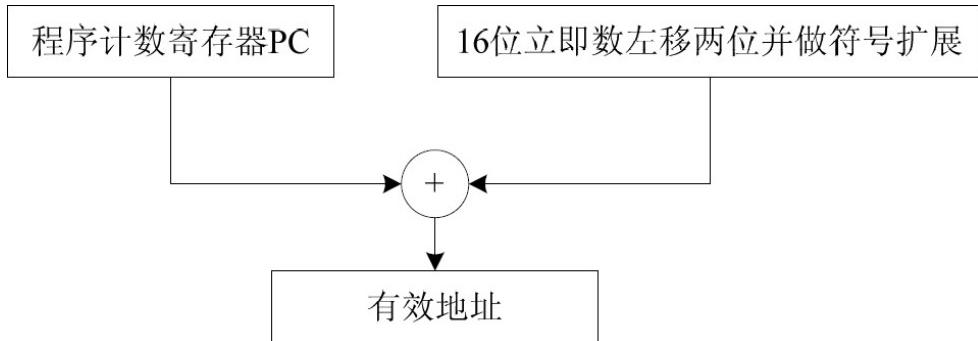


图1-7 PC相对寻址

1.4.7 协处理器CP0

协处理器一词通常用来表示处理器的一个可选部件，负责处理指令集的某个扩展，拥有与处理器相独立的寄存器。MIPS32架构提供了最多4个协处理器，分别是CP0~CP3。协处理器CP0用作系统控制，CP1、CP3用作浮点处理单元，而CP2被保留用于特定实现。除CP0外的协处理器都是可选的。

协处理器CP0的具体作用有：配置CPU工作状态、高速缓存控制、异常控制、存储管理单元控制等。CP0通过配置内部的一系列寄存器来完成上述工作。本书设计的处理器实现了CP0的部分功能，将在第10章详述。

1.4.8 异常

在处理器运行过程中，会从存储器中依次取出指令，然后执行，但是有一些事件会打断正常的程序执行流程，这些事件有中断（Interrupt）、陷阱（Trap）、系统调用（System Call）等，统称为异

常。异常发生后，处理器会转移到一个事先定义好的地址，在那个地址有异常处理例程，在其中进行异常处理，这个地址称为异常处理例程入口地址。异常处理完成后，使用异常返回指令`eret`，返回到异常发生前的状态继续执行。本书设计的处理器实现了对硬件复位、中断（包含软中断、硬中断）、`syscall`系统调用、无效指令、溢出、自陷6种异常的处理，将在第11章详述。

1.5 本书的目标与组织方式

本书的目标是实现一款兼容MIPS32指令集架构的处理器，命名为OpenMIPS，该处理器是通过使用硬件可编程语言Verilog HDL编写代码实现的，编写后的代码经过编译可以下载到FPGA芯片上，组成实际的硬件电路。计划实现两个版本——教学版OpenMIPS、实践版OpenMIPS。

教学版OpenMIPS处理器的主要设想是尽量简单，比如：在一个时钟周期内可以取到指令，完成存储、加载数据，这样处理器的运行情况（主要是流水线的运行）就比较理想化，与教科书相似，代码也很清晰简单，便于使用其进行教学、学术研究和讨论，也有助于学生理解课堂上讲授的知识。

实践版OpenMIPS处理器的设计目标是在教学版OpenMIPS的基础上实现Wishbone总线接口，这样就能将其挂接在Wishbone总线上，从而可以使用大量开源的SDRAM、Flash、GPIO、UART、LCD等模块的控制器，组成一个SOPC（System-On-a-Programmable-Chip，可编程片上系统），完成特定功能，成为一个能发挥实际作用的处理器。

全书组织如下。

为了方便读者理解，在第2章将对FPGA、Verilog HDL的基础知识做一介绍。

第3章介绍教学版OpenMIPS处理器的设计蓝图，包括设计目标、处理器接口，以及最终完成时，组成OpenMIPS的各个模块的作用，力图使读者有一个整体认识。并在本章详述了OpenMIPS处理器的实现方法。

随后的第4~11章，从最简单的情况开始，通过依次添加实现逻辑操作指令、移位操作指令、空指令、移动操作指令、算术操作指令、转移指令、加载存储指令、协处理器访问指令、异常相关指令，最终实现教学版OpenMIPS处理器。在每一类指令的实现过程中，都是先介绍该类指令的格式、作用、用法，然后介绍实现思路，接着通过修改代码实现该类指令，最后，编写测试程序，使用仿真的方式验证是否正确实现。

第12章实现了实践版OpenMIPS处理器。

第13章设计实现了基于实践版OpenMIPS的小型SOPC，该SOPC包括OpenMIPS处理器、SDRAM控制器、UART控制器、Flash控制器、GPIO模块、总线单元。

第14章将第13章实现的小型SOPC下载到实际的硬件平台上，编写测试程序，验证实践版OpenMIPS是否正确实现。

第15章介绍嵌入式实时操作系统μC/OS-II，并将其移植到本书设计的OpenMIPS处理器上，进一步验证了实践版OpenMIPS处理器实现

的正确性，也为OpenMIPS处理器发挥实际作用奠定了基础。

第2章 可编程逻辑器件与Verilog HDL

通过第1章的介绍，读者应该知道CPU内部有一些基本的电路，比如：译码电路、运算电路、控制电路，此外还有一些寄存器等。这些电路怎么实现呢？当然可以通过一大堆分立的元器件实现，实际上在2008年，美国加州的游戏开发人士Steve Chamberlin就自己制造了一款8位CPU，耗时18个月，花费1000美元，总共使用了1253条线缆，如图2-1所示，Steve Chamberlin为它起了一个十分贴切的名字——BMOW（Big Mess of Wires）。

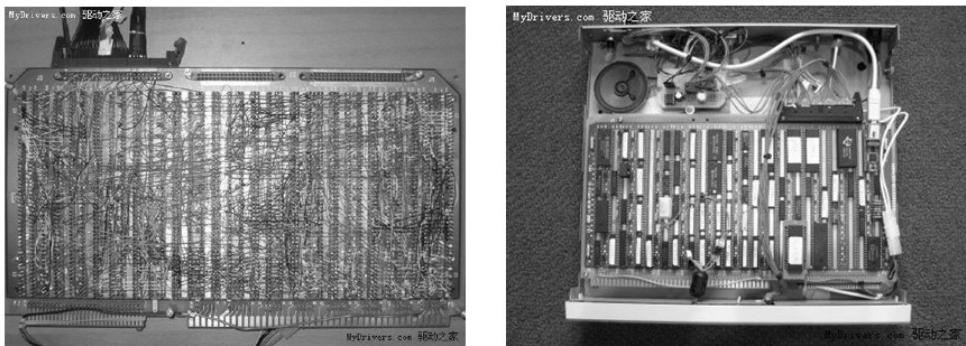


图2-1 BMOW的背面（左）与正面

还有一位叫Bill Buzbee的朋友也用200多块74系列的TTL集成电路纯手工制造了一款CPU。

上述事件只是证明了使用分立元件实现CPU的可行性，但那并不是实现CPU的好方法，本书是通过“代码+FPGA”的方式实现CPU的，本章将对其原理进行解释，并对使用的编程语言Verilog HDL进行介绍。

2.1 可编程逻辑器件概述

FPGA是可编程逻辑器件（Programmable Logic Device，PLD）的一种。PLD是上世纪70年代发展起来的一种新型器件，它的应用和发展不仅简化了电路设计，降低了开发成本，提高了系统可靠性，而且给数字系统的设计方法带来了革命性的变化。截止到现在，出现了多种工艺、不同原理的PLD，如下。

- PLA (Programmable Logic Array) 可编程逻辑阵列
- PAL (Programmable Array Logic) 可编程阵列逻辑
- GAL (Generic Array Logic) 通用阵列逻辑
- PROM (Programmable Read-Only Memory) 可编程只读存储器
- EPLD (Erasable Programmable Logic Device) 可擦除可编程逻辑器件
- CPLD (Complex Programmable Logic Device) 复杂可编程逻辑器件
- FPGA (Field Programmable Gate Array) 现场可编程门阵列

按照不同的内部结构可以将PLD器件分为如下两类。

1. 基于乘积项 (Product-Term) 结构的PLD器件

任何组合逻辑电路函数均可化为“与或”表达式，用“与门-或门”两级电路实现，而任何时序电路又都可以由组合电路加上存储元件（触发器）构成。因此，从原理上说，与或阵列加上触发器的结构就可以实现任意的数字逻辑电路。基于乘积项结构的PLD器件的主要结构就

是与或阵列，通过灵活配置的互连线，实现实意逻辑功能。其基本结构如图2-2所示。

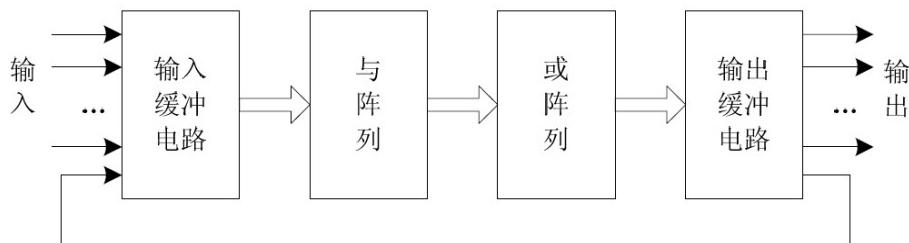


图2-2 基于乘积项结构的PLD器件结构图

基于乘积项结构的PLD器件由输入缓冲电路、与阵列、或阵列和输出缓冲电路四部分组成。“与阵列”和“或阵列”是主体，主要用来实现各种逻辑函数和逻辑功能；输入缓冲电路用于产生输入信号的原变量和反变量，并增强输入信号的驱动能力；输出缓冲电路主要用来对将要输出的信号进行处理，既能输出纯组合逻辑信号，也能输出时序逻辑信号。

PROM、PLA、PAL、GAL、EPLD和绝大部分CPLD器件都是采用乘积项（Product-Term）结构的PLD，内部基于与或阵列逻辑，这类器件多采用EEPROM或Flash工艺制作，掉电后不会丢失配置数据，器件规模一般小于5000门。

2. 基于查找表（Look-Up Table，LUT）结构的PLD器件

基于与或阵列的PLD器件的规模不容易做得很大，于是设计人员又开发出另外一种可编程逻辑器件，即查找表结构。其原理类似于ROM，物理结构基于静态存储器（MStatic RAM，SRA）和数据选择器（MUX），通过查表方式实现函数功能。函数值放在SRAM中，

SRAM的地址线即输入变量，不同的输入通过MUX找到对应的函数值并输出。N个输入项的逻辑函数可以由一个 2^N 位容量的SRAM实现。

图2-3是用2输入查找表实现2输入或门的示意图。2输入查找表中有4个存储单元，用来存储真值表中的4个值，输入变量A、B作为查找表中3个多路选择器的地址选择端，根据A、B值的组合从4个存储单元中选择一个作为查找表的输出，即实现了2输入或门的逻辑功能。

查找表结构的功能非常强，N个输入的查找表可以实现任意N个输入变量的组合逻辑函数。从理论上讲，只要能够增加输入信号线和扩大存储器容量，用查找表就可以实现任意输入变量的函数。但在实际应用中，查找表的规模受技术和成本因素的限制。每增加一个输入变量，查找表SRAM的容量就要扩大一倍，SRAM的容量与输入变量数N的关系是 2^N 倍。8输入变量的查找表需要256b容量的SRAM，而16个输入变量的查找表则需要64Kb容量的SRAM，这个规模已无法忍受。实际上，FPGA器件查找表的输入变量一般不超过5个，多于5个输入变量的逻辑函数可由多个查找表通过组合或级联实现。

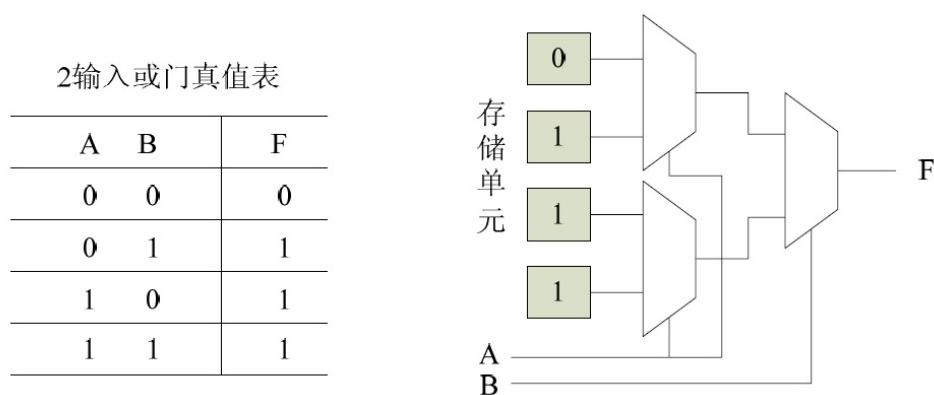


图2-3 用2输入查找表实现或门功能

绝大多数FPGA器件都是基于SRAM查找表结构实现的。特点是集成度高（可实现百万逻辑门以上设计规模）、逻辑功能强，可实现大规模的数字系统设计和复杂的算法运算，但掉电后会丢失配置数据，需外挂非易失配置器件以存储配置数据，才能构成可独立运行的系统。在FPGA内部一般还会集成更多的逻辑功能块，如存储器块、DSP块、硬件乘法器、数字锁相环等，用以满足数字信号处理、数字通信等应用的需要。

本书最终实现的实践版OpenMIPS处理器就将下载到FPGA上运行，使用的是Altera公司的EP2C35系列的FPGA，其具有33216个LE（Logic Element），每个LE主要由一个4输入查找表和一个可编程的寄存器构成。

2.2 基于PLD的数字系统设计流程

PLD不仅是技术的革新，也带来观念的革新、设计流程的革新，基于PLD的数字系统设计流程如图2-4所示。本节将分别介绍流程中的各个阶段。

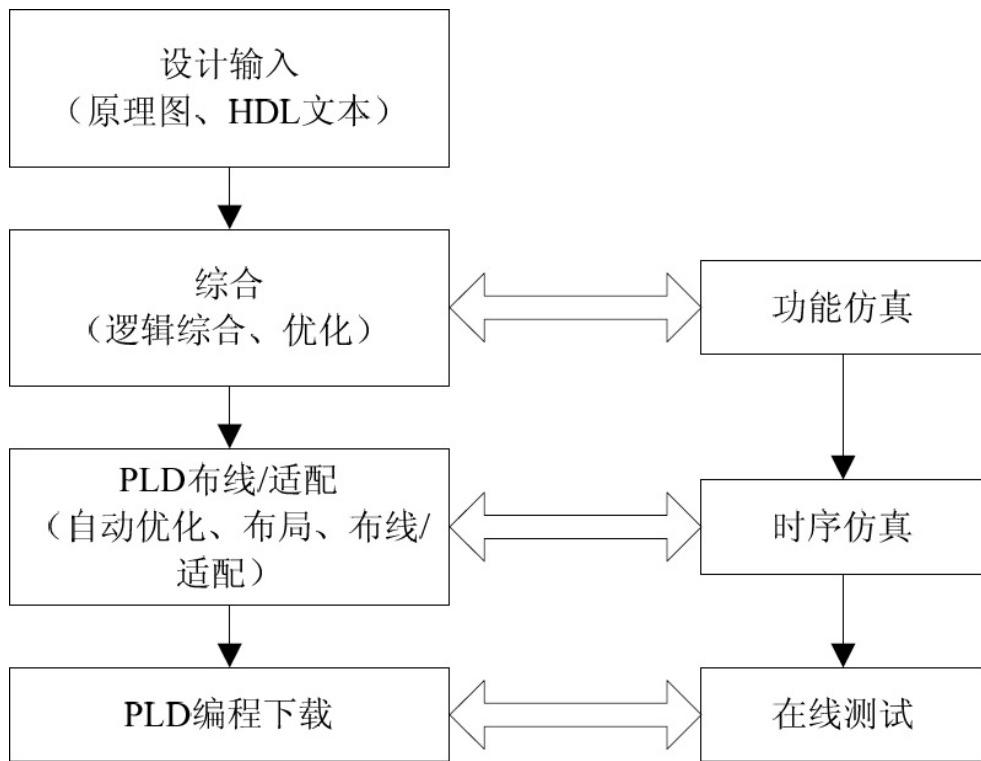


图2-4 基于PLD的数字系统设计流程

2.2.1 设计输入

设计输入是将设计者所设计的电路以开发软件要求的某种形式表达出来，并输入到相应软件中的过程。设计输入有多种方式，最常用的是原理图方式和HDL文本方式两种。

1. 原理图输入

原理图（Schematic）是图形化的表达方式，使用元件符号和连线来描述设计。原理图输入对用户来说很直观，尤其对于表现层次结构、模块化结构更为方便，适合描述连接关系和接口关系，而描述逻辑功能则比较烦琐。其要求设计工具提供必要的元件库或逻辑宏单元。如果输入的是较为复杂的逻辑或者元件库中不存在的模型，采用

原理图输入的方式往往很不方便，此外，原理图方式的设计可重用性、可移植性也差一些。

图2-5是使用原理图输入的二选一选择器的电路。有三个输入：a0、a1、s，一个输出y，当s为1时，y的值等于a1的值，当s为0时，y的值等于a0的值。

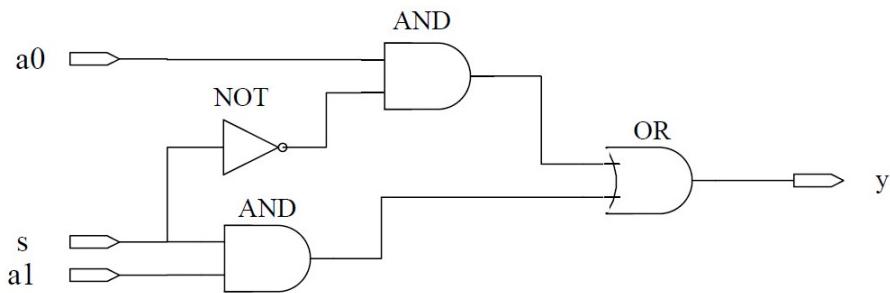


图2-5 二选一选择器

2. HDL文本输入

硬件描述语言（Hardware Description Language, HDL）是一种用文本形式来描述和设计电路的语言。设计者可利用HDL语言来描述自己的设计，然后利用相应的工具进行综合，变为某种目标文件，最后下载到PLD器件，实现具体电路。目前常用的HDL有VHDL、Verilog HDL等。

VHDL 和 Verilog HDL 各有优点，可以进行算法级（Algorithm Levels）、寄存器传输级（RTL）、门级（Gate Levels）等各种层次的逻辑设计，也可以进行仿真验证、时序分析等。由于HDL语言的标准化，易于将设计移植到不同厂家的芯片中，信号参数也容易改变和修改。此外，采用HDL进行设计还具有工艺无关性，使得设计人员在功能设计、逻辑验证阶段可以不必过多考虑门级及工艺实现的具体细

节，只需根据系统设计的要求，施加不同的约束条件，即可设计出实际的电路。如下是使用Verilog HDL实现的二选一选择器的代码。

```
module mux2(a0, a1, s, y);
    input s, a0, a1;
    output y;
    assign y=s ? a1:a0;
endmodule
```

本书使用Verilog HDL实现OpenMIPS处理器。

2.2.2 综合

综合（Synthesis）是将较高级抽象层次的设计描述自动转化为较低层次描述的过程。有以下几种综合形式。

- 将算法表示、行为描述转换到寄存器传输级（RTL），即从行为描述到结构描述。
- 将RTL级描述转换到逻辑门级，称为逻辑综合。
- 将逻辑门表示转换到PLD器件的配置网表表示，有了配置网表就可完成基于PLD器件的系统实现。

综合器就是能够自动实现上述转换的软件工具，其能够将原理图或HDL语言表达、描述的电路编译成由与或阵列、RAM、触发器、寄存器等逻辑单元组成的电路网表。

2.2.3 布局布线

布局布线可以理解为将综合生成的电路网表映射到具体的目标PLD器件，并产生最终可下载文件的过程。布局布线将综合后的电路网表针对某一具体的目标PLD器件进行逻辑映射，把整个设计分为多个适合PLD器件内部逻辑资源实现的逻辑小块，并根据用户的设定在速度和面积之间做出选择或折中。其中布局是将已分割的逻辑小块放到PLD器件内部逻辑资源的具体位置，并使它们易于连线；布线则是利用PLD器件的布线资源完成各功能块之间、反馈信号之间的连接。

布局布线完成后产生如下一些重要的文件。

- (1) 芯片资源耗用的情况报告。
- (2) 产生延时网表结构，以便于进行精确的时序仿真，能比较精确地预测未来芯片的实际性能。
- (3) 器件编程文件，如用于CPLD编程的JEDEC、POF等格式的文件；用于FPGA配置的SOF、JAM、BIT等格式的文件。

2.2.4 下载

把布局布线过程中产生的器件编程文件放入PLD的过程称为下载。通常将对CPLD器件的下载称为编程，将对FPGA器件的下载称为配置（Configuration）。下载后，PLD内部的与或门（对FPGA而言就是查找表）会按照编程文件的要求变化，从而实现了设计的电路。

2.2.5 仿真

从图2-4中可发现其中有仿真环节。仿真（Simulation）也称为模拟，是对所设计电路的功能进行检验。用户可以在设计过程中对整个系统和各个模块进行仿真，即在计算机上用软件验证功能是否正确，各部分的时序配合是否准确。如果有问题可以随时进行修改，从而避免了逻辑错误。规模越大的设计，越需要进行仿真。

仿真包括功能仿真和时序仿真。不考虑信号时延等因素的仿真，称为功能仿真，又叫前仿真；时序仿真又称后仿真，它是在选择具体器件并完成布局布线后进行的包含时延的仿真。由于不同器件的内部时延不一样，不同的布局、布线方案也会影响时延，因此在设计实现中，对网络和逻辑块进行时序仿真，分析定时关系，估计设计性能是非常必要的。

本书实现的教学版OpenMIPS处理器就是主要通过仿真来验证其实现是否正确，只有实践版OpenMIPS才配置到具体的FPGA芯片中。

2.2.6 工具介绍

在基于PLD的数字系统设计流程的每一个阶段都有相应的工具支持，有些工具是集成的，可以完成整个设计流程的各个阶段，有些工具是专门针对某一设计阶段的。本书在设计实现OpenMIPS处理器时使用的工具如下。

- 设计输入工具：UltraEdit
- 仿真工具：ModelSim
- 集成工具：QuartusII

因为实践版OpenMIPS是下载到Altera公司的FPGA芯片中，所以集成工具使用的是Altera公司的QuartusII。一般而言，集成工具最好选择目标PLD芯片厂商提供的工具，因为厂商的工具会针对自己器件的工艺特点做优化设计，从而提高资源利用率、降低功耗、改善性能。

2.3 Verilog HDL简介

本书实现的OpenMIPS处理器是使用Verilog HDL编写的，所以本章接下来的几节将介绍Verilog HDL的一些基本知识，包括语法、结构等。因为本书并不是一本讲授Verilog HDL的专门书籍，所以此处介绍的内容并不是Verilog HDL的全部，只是一些基础知识，以及在OpenMIPS处理器实现过程中会使用到的知识。读者如果对Verilog HDL有进一步了解的需求，可以参考相关书籍，这方面有许多非常优秀的书籍。笔者推荐《数字系统设计与Verilog HDL（第4版）》，本章关于Verilog HDL的介绍也部分参考了该书。

Verilog HDL由GDA（Gateway Design Automation）公司的Phil Moorby于1983年首创，之后，Moorby又设计了Verilog-XL仿真器，Verilog-XL仿真器的大获成功，也使得Verilog HDL得到了推广使用。1989年，Cadence收购了GDA，1990年，Cadence公开发布了Verilog HDL，并成立了OVI（Open Verilog International）组织，专门负责Verilog HDL的发展。由于Verilog HDL具有简洁、高效、易用、功能强等优点，已逐渐为众多设计者所接受和喜爱，其发展经历了几个关键节点。

- 1995年，Verilog HDL成为IEEE标准，称为IEEE Standard 1364-1995（Verilog-1995）。

- 2001年，IEEE Standard 1364-2001（Verilog-2001）获得通过，其对Verilog-1995做了扩充和增强。另外，修改了一些语法结构，使之更易于使用。
- 2002年，为了使综合器输出的结果和基于IEEE Standard 1364-2001的仿真和分析工具的结果相一致，推出了IEEE 1364[1].1-2002标准，其对Verilog HDL的RTL级综合定义了一系列的建模准则。
- 2005年，Verilog HDL再次进行了更新，即IEEE Standard 1364-2005（Verilog-2005）。该版本只是对上一版本的细微修正。这个版本还包括了一个相对独立的新部分，即Verilog-AMS（Analog and Mixed-Signal），这个扩展使得传统的Verilog HDL可以对集成模拟和混合信号的系统进行建模。

Verilog HDL具有下述特点。

- (1) Verilog HDL是在C语言的基础上发展而来的，就语法结构而言，Verilog HDL继承了C语言的很多语法结构，两者有许多相似之处。
- (2) 既适于可综合的电路设计，也可胜任电路与系统的仿真。
- (3) 能在多个层次上对所设计的系统加以描述，从开关级、门级、寄存器传输级（RTL）到行为级，都可以胜任，同时Verilog HDL不对设计规模施加任何限制。
- (4) 灵活多样的电路描述风格，可进行行为描述，也可进行结构描述；支持混合建模，一个设计中的各个模块可以在不同的设计层次上建模和描述。

(5) 内置多种基本逻辑门，如and、or和nand等，可方便地进行门级结构描述；内置多种开关级元件，如pmos、nmos和cmos等，可进行开关级的建模。

(6) 用户定义原语（UDP）创建的灵活性。用户定义的原语既可以是组合逻辑，也可以是时序逻辑；通过编程语言接口（PLI）机制可进一步扩展Verilog HDL语言的描述能力。

2.4 Verilog HDL中模块的结构

Verilog程序的基本设计单元是“模块”（Module），一个模块有其特定的结构，如图2-6所示。Verilog的模块完全定义在module与endmodule关键字之间，每个模块包括四个主要部分：模块声明、端口定义、数据类型说明和逻辑功能描述。

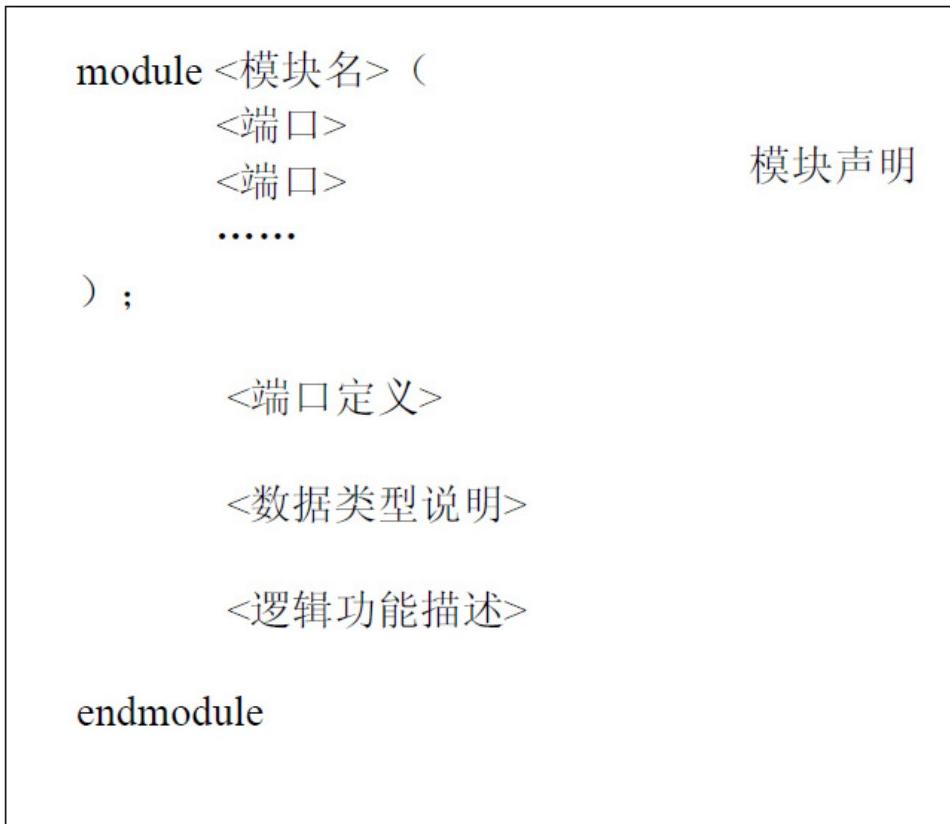


图2-6 Module的基本结构

如下是一个实现32位加法器的模块。有两个输入信号in1、in2，两者相加的结果通过out输出。

```

module add32(in1, in2, out);           // 模块声明

    input   in1, in2;                  // 端口定义，此处是输入端口
    output  out;                     // 端口定义，此处是输出端
    口

    wire[31:0] in1, in2, out;        // 数据类型说明，此处都是
    wire型

```

```
assign out = in1 + in2;           // 逻辑功能描述  
  
endmodule
```

下面结合该加法器的例子，对Module的基本结构进行说明。

1. 模块声明

模块声明包括模块名字，以及输入、输出端口列表，其格式如下。

```
module 模块名(端口1, 端口2, 端口3...);
```

2. 端口定义

明确说明模块端口的方向（输入、输出、双向等），其格式如下。

```
input 端口1, 端口2, 端口3 ...;          // 输入端口  
output 端口1, 端口2, 端口3 ...;         // 输出端口  
inout 端口1, 端口2, 端口3 ...;          // 双向端口
```

3. 数据类型说明

对模块中所有用到的信号（包括端口信号、节点信号等）都必须进行数据类型的定义。Verilog HDL提供了各种信号类型，下面是几种定义信号类型的例子。各数据类型的具体含义将在2.5.2节详述。

```
reg a;                                // 定义信号a的数据类型为reg型  
wire[31:0] out ;                      // 定义信号out的数据类型为32位wire型
```

对于端口，可以将数据类型说明与端口定义放在一条语句中完成，于是，上文的32位加法器可以改为如下形式。

```
module add32(in1, in2, out);

    input wire[31:0] in1, in2;      // 将端口定义与类型说明放在一条语句中

    output wire[31:0] out;

    assign out = in1 + in2;

endmodule
```

对于端口，还可以将端口定义、数据类型说明都放在模块声明中，而不再放在模块内部，于是，上文的32位加法器还可以改为如下形式，更为简便。

```
// 将端口定义、数据类型说明放在模块声明中

module add32(input wire[31:0] in1,
              input wire[31:0] in2,
              output wire[31:0] out);

    assign out = in1 + in2;

endmodule
```

4. 逻辑功能描述

模块中最核心的部分就是逻辑功能描述，可以有多种方法在模块中描述和定义逻辑功能。几种基本方法如下，具体内容将在2.6节详述。

- 用assign持续赋值语句定义
- 用always过程块定义
- 调用元件（也称为元件例化）

2.5 Verilog HDL基本要素

2.5.1 常量

Verilog中的常量（Constant）有三种：整数、实数、字符串。在OpenMIPS的实现过程中只使用了整数常量，所以，此处也仅介绍整数常量。其格式如图2-7所示。

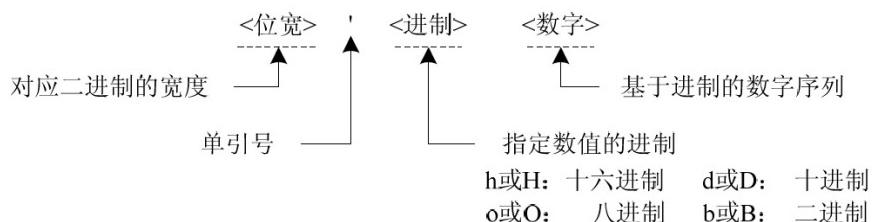


图2-7 整数常量的格式

一些整数常数的例子如下。

```
8'b11000101      // 宽度为8位的二进制数，数值为11000101，等于十进制的197  
8'h8a            // 宽度为8位的十六进制数，数值为8a，等于十进制的138
```

```
5'o27          // 宽度为5位的八进制数，数值为27，等于十进制的23  
4'd10          // 宽度为4位的十进制数，数值为10
```

如果没有明确指明进制，那么默认是十进制数。

2.5.2 变量声明与数据类型

Verilog中变量声明的格式如图2-8所示。只有数据类型、变量名是必要的，其他部分都可以省略。如果省略符号和位宽，那么根据数据类型设置为默认值。如果省略元素数，那么默认声明元素数为1。

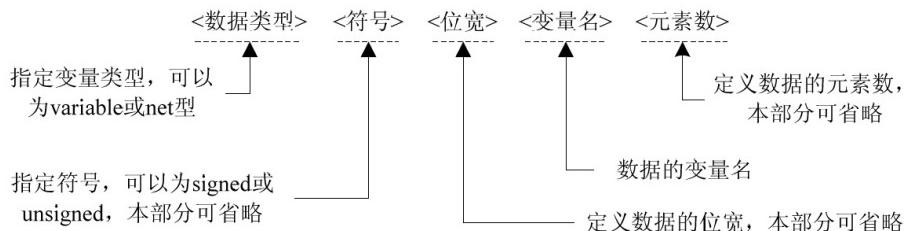


图2-8 变量声明的格式

数据类型可以是net型、variable型，分别介绍如下。

1. net型变量

net型相当于硬件电路中各种物理连接，其特点是输出的值紧跟输入值的变化而变化。net型变量包括多种类型，如表2-1所示。

表2-1 net型变量

名称	默认位宽	默认符号	含义

wire、 tri	1位	无符号	线连接
wor、 trior	1位	无符号	线或连接
wand、 triand	1位	无符号	线与连接
tri1、 tri0	1位	无符号	有上拉或下拉的连接
supply0、 supply1	1位	无符号	接地或接电源的连接

本书在实现OpenMIPS处理器的时候只使用了其中的wire类型。wire是最常用的net型变量，Verilog HDL模块中的输入、输出信号在没有明确指定数据类型时，都被默认为wire型。wire型信号可以用作任何表达式的输入，也可以用作assign语句和实例元件的输出，如前文中的32位加法器对out信号的赋值。对于综合器而言，wire型变量的取值可为0、1、X、Z，其中0表示低电平、逻辑0；1表示高电平、逻辑1；X表示不确定或未知的逻辑状态；Z表示高阻态。如果没有赋值，默认为高阻态Z。

2. variable型变量

variable型变量是可以保存上次写入数据的数据类型，一般对应硬件上的一个触发器或锁存器等存储元件，但并不是绝对的，在综合器综合的时候，会根据其被赋值的情况来具体确定是映射成连线还是映射为存储元件。variable型变量也包括多种类型，如表2-2所示。本书在实现OpenMIPS处理器的时候只使用了其中的reg类型。

表2-2 variable型变量

名称	默认位宽	默认符号	含义
reg	1位	无符号	比特数据
integer	32位	无符号	整数
real	64位	无符号	实数

variable型变量必须在过程语句（initial或always）中实现赋值，这种赋值方式称为过程赋值，将在2.6节详述。

2.5.3 向量

图2-8变量声明格式中的位宽如果为1，那么对应的变量为标量，如果不为1，那么对应的变量为向量，默认为标量。向量的位宽用下面的形式定义。

[MSB : LSB]

冒号左边的数字表示向量的最高有效位MSB（Most Significant Bit），冒号右边的数字表示向量的最低有效位LSB（Least Significant Bit）。例如。

```
wire [3:0] bus;      // 4位的wire型向量bus，其中bus[3]是最高位，  
bus[0]是最低位  
reg [31:5] ra;      // 27位的reg型向量ra，其中ra[31]是最高位，
```

ra[5]是最低位

```
reg [0:7] rc; // 8位的reg型向量rc，其中rc[0]是最高位，rc[7]是最低位
```

向量有两种，一种是向量类向量，另一种是标量类向量，可以使用关键字区分，如下。

```
wire vectored [7:0] databus; // 使用关键字vectored，表示是向量类向量  
reg scalared [31:0] rega; // 使用关键字scalared，表示是标量类向量
```

如果没有明确指出，那么默认是标量类向量。本书也只用到了标量类向量，对标量类向量可以任意选中其中的一位或相邻几位，分别称为位选择（bit-select）和域选择（part-select）。例如。

```
A = rega[6]; // 位选择，将向量rega的其中一位赋值给变量A  
B = rega[5:2]; // 域选择，将向量rega的第5、4、3、2位赋值给变量B
```

在OpenMIPS的实现过程中，使用了存储器，存储器可看作是二维的向量。如下就是一个存储器的定义，定义了一个深度为64，每个元素宽度为32bit的存储器。

```
reg [31:0] mem[63:0]; // mem是深度为64，字长为32bit的存储器
```

2.5.4 运算符

Verilog HDL中定义的运算符包括：算术运算符、逻辑运算符、位运算符、关系运算符、等式运算符、缩位运算符、移位运算符、条件运算符和位拼接运算符。详情如表2-3所示。

表2-3 Verilog HDL中定义的运算符

运算符种类	运算符	含义
算术运算符	+	加法
算术运算符	-	减法
	*	乘法
	/	除法
	%	求余
	&&	逻辑与
逻辑运算符		逻辑或
	!	逻辑非
	~	按位取反
位运算符	&	按位与
		按位或
	^	按位异或

	$\sim \wedge$ 或 $\wedge \sim$	按位同或
关系运算符	<	小于
	\leq	小于等于
	>	大于
	\geq	大于等于
等式运算符	$=$	等于
	\neq	不等于
	\equiv	全等
\neq	不全等	
缩位运算符	$\&$	与
	$\sim \&$	与非
	$ $	或
	$\sim $	或非
	\wedge	异或
	$\sim \wedge$ 或 $\wedge \sim$	同或
移位运算符	$>>$	右移

	<<	左移
条件运算符	? :	条件运算
位拼接运算符	{ }	拼接

表2-3中的大部分运算符都很好理解，本书不再详释，只做如下几点说明。

(1) 等式运算符中的“==”与“===”的区别是：对于“==”运算，参与比较的两个操作数必须逐位相等，其结果才为1，如果某些值是不定态X或高阻态Z，那么得到的结果是不定值X；而对于“===”运算，则要求对参与运算的操作数中为不定态X或高阻态Z的位也进行比较，两个操作数必须完全一致，其结果才为1，否则结果为0。例如。

```
reg [4:0] a = 5'b11x01;
reg [4:0] b = 5'b11x01;
```

针对上面的a、b，“a==b”的返回结果为X，而“a==b”的返回结果为1。

(2) 缩位运算符与位运算的运算符号、逻辑运算法则都是一样的，但是缩位运算符是对单个操作数进行与、或、异或的递推运算，它放在操作数的前面，能够将一个矢量减为一个标量。例如。

```
reg [3:0] a;
b = &a; // 等效于b = ((a[0] & a[1]) & a[2]) & a[3]
```

而位运算需要对两个操作数按对应位分别进行逻辑运算，例如。

```
wire [3:0] a = 4'b0011;  
wire [3:0] b = 4'b0101;  
那么a&b = 4'b0001, a|b = 4'b0111
```

(3) 位拼接运算符：用来将两个或多个信号的某些位拼接起来。其格式如下。

```
{比特序列0, 比特序列1,..... }
```

例如，在进行加法运算时，可将和与进位输出拼接在一起使用。

```
input [3:0] ina,inb;           // 加法输入  
output [3:0] sum;             // 加法的和  
output cout;                 // 进位  
assign {cout, sum} = ina + inb; // 将和与进位拼接在一起
```

位拼接还可以用来重复信号的某些位，其格式如下。

```
{重复次数{被重复数据}}
```

利用上面的功能，可以实现对信号的符号扩展，例如。

```
// 将 Data 的符号位进行扩展, s_data =  
{Data[7],Data[7],Data[7],Data[7],Data}  
wire [7:0] Data;  
wire [11:0] s_data;  
s_data = {{4{Data[7]}},Data};
```

(4) 运算符的优先级如图2-9所示。

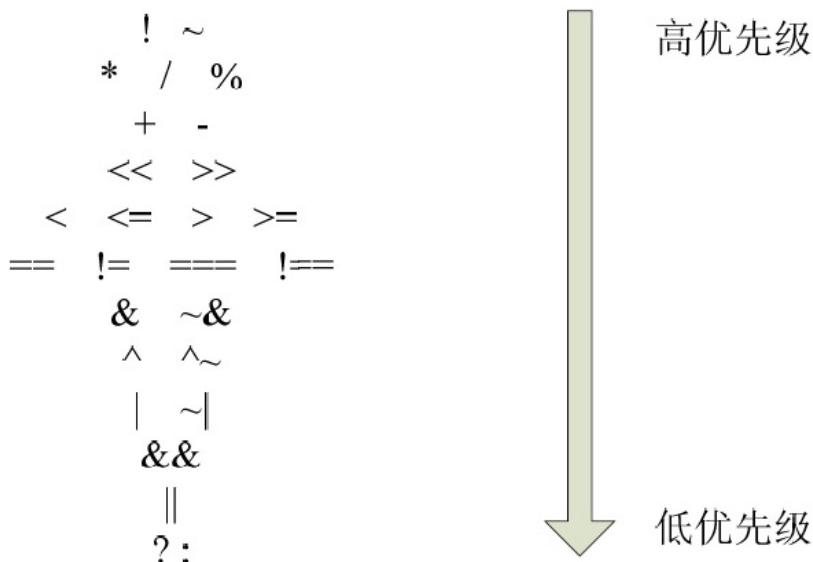


图2-9 运算符的优先级

2.6 Verilog HDL行为语句

2.6.1 过程语句

Verilog定义的模块一般包括过程语句，过程语句有两种：initial、always。其中initial常用于仿真中的初始化，其中的语句只执行一次，而always中语句则是不断重复执行的。此外， always过程语句是可综合的， initial过程语句是不可综合的。

1. always过程语句

always过程语句的格式如图2-10所示。

```
always @ (<敏感信号表达式>)
begin
    // 语句序列
end
```

图2-10 always过程语句的格式

always过程语句通常是带有触发条件的，触发条件写在敏感信号表达式中，敏感信号表达式又称为事件表达式或敏感信号列表，当该表达式中变量的值改变时，就会引发其中语句序列的执行。因此，敏感信号表达式中应列出影响块内取值的所有信号。

(1) 敏感信号表达式的格式

如果有两个或两个以上的敏感信号时，它们之间使用“or”连接，此处还是以32位加法器为例，2.4节是使用assign直接赋值的，其实也可以使用always过程语句实现。只要被加数in1、加数in2中的任何一个改变，都会触发always过程语句，在其中进行加法运算。这里有两个敏感信号in1、in2，使用“or”连接。

```
module add32(input wire[31:0] in1,
              input wire[31:0] in2,
              output reg[31:0] out);

always @ (in1 or in2)          // 使用always过程语句实现加法
begin
    out = in1 + in2;
end
endmodule
```

```
begin
    out = in1 + in2;
end

endmodule
```

敏感信号列表中的多个信号可以使用逗号隔开，上面的32位加法器可以修改为如下形式。

```
module add32(input wire[31:0] in1,
              input wire[31:0] in2,
              output reg[31:0] out);

always @ (in1, in2)          //多个敏感信号使用逗号分隔

begin
    out = in1 + in2;
end

endmodule
```

敏感信号列表也可以使用通配符“*”，表示在该过程语句中的所有输入信号变量，上面的32位加法器可以修改为如下形式。

```
module add32(input wire[31:0] in1,
              input wire[31:0] in2,
              output reg[31:0] out);

always @ (*) //使用通配符表示过程语句中的所有输入信号
变量

begin
    out = in1 + in2;
end

endmodule
```

(2) 组合电路与时序电路

敏感信号可以分为两种类型：一种为电平敏感型，另一种为边沿敏感型。前一种一般对应组合电路，如上面给出的加法器的例子，后一种一般对应时序电路。对于时序电路，敏感信号通常是时钟信号，Verilog HDL提供了posedge、negedge两个关键字来描述时钟信号。posedge表示以时钟信号的上升沿作为触发条件，negedge表示以时钟

信号的下降沿作为触发条件。还是以32位加法器为例，可以为其添加一个时钟同步信号，如下。

```
module add32(input wire          clk, //增加了一个时钟输入信号
              input wire[31:0]  in1,
              input wire[31:0]  in2,
              output reg[31:0]  out);

    always @ (posedge clk)           //在时钟信号的上升沿会触发
always中的语句
begin
    out = in1 + in2;
end

endmodule
```

在时钟信号的上升沿，才会进行加法运算，这一点与前面的加法器不同，也就是当被加数in1、加数in2变化时，并不会立即改变输出out，而是要等待时钟信号的上升沿。

2. initial过程语句

initial过程语句的格式如图2-11所示。

```
initial  
begin  
  
    // 语句序列  
  
end
```

图2-11 initial过程语句的格式

initial过程语句不带触发条件，并且其中的语句序列只执行一次。initial过程语句通常用于仿真模块中对激励向量的描述，或用于给寄存器赋初值，它是面向模拟仿真过程语句，通常不能被综合。如下是initial过程语句的一个例子，用于给存储器mem赋初值。

```
initial  
begin  
    for(addr = 0; addr < size; addr = addr+1) // for是一种循环语  
句，下文会介绍  
    mem[addr] = 0;  
end
```

2.6.2 赋值语句

赋值语句有两种：持续赋值语句、过程赋值语句。

1. 持续赋值语句

assign为持续赋值语句，主要用于对wire型变量的赋值。如上文中加法器的例子。

2. 过程赋值语句

在always、initial过程中的赋值语句称为过程赋值语句，多用于对reg型变量进行赋值，分为非阻塞赋值和阻塞赋值两种方式。

(1) 非阻塞赋值 (Non-Blocking)

赋值符号为“<=”，例如。

```
b <= a
```

非阻塞赋值在整个过程语句结束时才会完成赋值操作，即b的值并不是立刻改变的。

(2) 阻塞赋值 (Blocking)

赋值符号为“=”，例如。

```
b = a
```

阻塞赋值在该语句结束时就立即完成赋值操作，即b的值在这条语句结束后立刻改变。如果在一个块语句中，有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句就不能被执行，仿佛被阻塞了一样，因此称为阻塞赋值方式。

在always过程块中，阻塞赋值可以理解为赋值语句是顺序执行的，而非阻塞赋值可以理解为赋值语句是并发执行的。如图2-12所

示。在一个过程块中，阻塞式赋值与非阻塞式赋值只能使用其中一种。

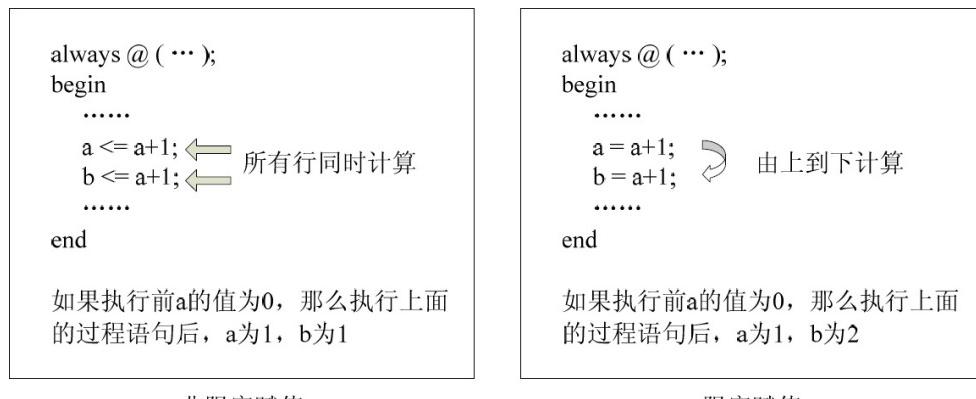


图2-12 非阻塞赋值与阻塞赋值

2.6.3 条件语句

条件语句有if-else、case两种，应放在always块内。分别介绍如下。

1. if-else语句

if-else语句的格式有如下三种。

(1) if(表达式)
语句
语句序列1; // 非完整性IF

(2) if(表达式)
IF语句
else
语句序列1; // 二重选择的
语句序列2;

```
(3) if(表达式1)          语句序列1;           // 多重选择的
IF语句
    else if(表达式2)      语句序列2;
    else if(表达式3)      语句序列3;
    .....
    else if(表达式n)      语句序列n;
else                      语句序列n+1;
```

上述格式中的“表达式”一般为逻辑表达式或关系表达式，也可能是1位的变量。系统对表达式的值进行判断，如果为0、X、Z，则按“假”处理，如果为1，则按“真”处理。语句序列可以是单句，也可以是多句，多句时需使用begin-end块语句括起来。

还是以32位加法器为例，为其添加一个复位信号rst，如果rst为高电平，那么复位信号有效，输出out为0，反之，复位信号无效，输出out为两个输入信号之和。

```
module add32(input wire          rst,           // 增加了一个复位信号
              input wire[31:0]  in1,
              input wire[31:0]  in2,
              output reg[31:0]  out);

always @ (*)
begin
    if(rst == 1'b1)
        out <= 32'h0;           // 如果复位信号有效，那么输出out为0
    else
        out <= in1 + in2;       // 反之，输出out为两个输入信号之和
end
```

```
end  
  
endmodule
```

2. case语句

相对于if-else语句只有两个分支而言，case语句是一种多分支语句，所以case语句多用于多条件译码电路，如译码器、数据选择器、状态机及微处理器的指令译码等。其格式如下。

```
case(敏感表达式)  
    值1: 语句序列1;  
    值2: 语句序列2;  
    .....  
    值n: 语句序列n;  
    default: 语句序列n+1;  
endcase
```

当敏感表达式的值等于“值1”时，执行语句序列1；当等于“值2”时，执行语句序列2；依次类推。如果敏感表达式的值与上面列出的值都不符，那么执行default后面的语句序列。如下代码是一个简单的运算单元，可执行加法或减法运算，如果输入变量type的值为1，那么执行加法运算，如果type的值为0，那么执行减法运算。

```
module add_sub32(input wire type,      // type决定运算类型  
                  input wire[31:0] in1,  
                  input wire[31:0] in2,  
                  output reg[31:0] out);
```

```

always @ (*)
begin
    case(type)
        1'b1 : out <= in1 + in2;      // type为1, 执行加法运算
        1'b0 : out <= in1 - in2;      // type为0, 执行减法运算
        default : out <= 32'h0;
    endcase
end

endmodule

```

case语句中，敏感表达式与值1~n之间的比较是一种全等比较，必须保证两者的对应位全等。casez、casex语句是case语句的两种扩展。

- 在casez语句中，如果比较的双方某些位的值为高阻Z，那么对这些位的比较结果就不予考虑，只需考虑其他位的比较结果。
- 在casex语句中，如果比较的双方某些位的值为Z或X，那么对这些位的比较结果就不予考虑，只需考虑其他位的比较结果。

此外，还有一种表示X或Z的方式，即用表示无关值的符号“?”来表示，例如。

```

case(a)
    2'b1x : out <= 1;      //只有a等于2'b1x时, out才等于1

```

```
casez(a)
 2'b1x : out <= 1;      //a等于2'b1x、2'b1z时，out等于1

casex(a)
 2'b1x : out <= 1;      //a等于2'b10、2'b11、2'b1x、2'b1z时，out等
于1

case(a)
 2'b1? : out <= 1;      //a等于2'b10、2'b11、2'b1x、2'b1z时，out等
于1
```

2.6.4 循环语句

Verilog HDL中存在四种类型的循环语句：for、forever、repeat、while，用来控制语句的执行次数，分别介绍如下。

1. for语句

for语句的格式如下。这与C语言是相似的。

```
for(循环变量赋初值；循环结束条件；修改循环变量)
  执行语句序列；
```

一个使用for语句实现7人表决器的例子如下。通过for循环统计赞成的人数，若超过4人（含4人）赞成则通过，其中vote[7:1]表示7个人

的投票情况，vote[i]为1，表示第i个人投的是赞成票，反之是反对票，pass是输出，超过4个人赞成，pass为1，反之为0。

```
module vote7(vote, pass);

    input wire[7:1] vote;
    output reg      pass;
    reg[2:0]        sum;
    integer         i;

    always @ (vote)
    begin
        sum = 0;
        for(i=1; i<7; i=i+1)
            if(vote[i])
                sum = sum+1;      //如果vote[i]为1，那么sum加1，注意此处使用阻塞赋值
        if(sum[2] == 1'b1) //如果sum大于等于4，那么输出pass为1
            pass = 1;
        else
            pass = 0;
    end

endmodule
```

2. forever语句

forever语句的格式如下。

```
forever begin
```

语句序列

```
end
```

forever循环语句连续不断地执行其中的语句序列，常用来产生周期性的波形。在2.8节编写仿真用的Test Bench文件时，会给出forever语句的例子。

3. repeat语句

repeat语句的格式如下。

```
repeat(循环次数表达式) begin
```

语句序列

```
end
```

4. while语句

while语句的格式如下。

```
while(循环执行条件表达式) begin
```

语句序列

```
end
```

while语句在执行时，首先判断循环执行条件表达式是否为真，若为真，则执行其中的语句序列，然后再次判断循环执行条件表达式是否为真，若为真，则再次执行其中的语句序列，如此反复，直到循环执行条件表达式不为真为止。

2.6.5 编译指示语句

Verilog HDL和C语言一样提供了编译指示功能，允许在程序中使用编译指示（Compiler Directives）语句，在编译时，通常先对这些指示语句进行预处理，然后再将预处理的结果和源程序一起进行编译。

编译指示语句以“`”开始，以区别其他语句。常用的编译指示语句有：`define、`include、`ifdef、`else、`endif，分别介绍如下。

1. 宏替换`define

`define可以用一个简单的名字或有意义的标识（也称为宏名）代替一个复杂的名字或变量，其格式如下。

```
`define 宏名 变量或名字
```

例如：一般在时序电路中会有一个复位信号，当该复位信号为高电平时表示复位信号有效，当该复位信号为低电平时，表示复位信号无效。分别执行不同的代码，如下。

```
always @ (clk)
begin
  if(rst == 1'b1)
```

```
//复位有效  
else  
    //复位无效  
end
```

一种更为友好的书写方式，是使用宏定义，如下。

```
// 定义宏RstEnable表示复位信号有效，这个名字对读者而言更有意义  
`define RstEnable 1'b1  
  
.....  
  
always @ (clk)  
begin  
    if(rst == `RstEnable) // 在编译的时候会自动将`RstEnable替换成  
    1'b1  
        //复位有效  
    else  
        //复位无效  
end
```

2. `include语句

`include是文件包含语句，它可将一个文件全部包含到另一个文件中，使用格式如下。

```
`include "文件名"
```

在OpenMIPS处理器的实现过程中，我们定义了很多宏，这些宏都集中在文件defines.v中，如果某一程序需要使用其中的宏定义，就可以在程序文件的开始使用`include语句将defines.v文件包含进来即可，如下。

```
`include "defines.v"
```

3. 条件编译语句`ifdef、`else、`endif

条件编译语句`ifdef、`else、`endif可以指定仅对程序中的部分内容进行编译，有两种使用形式。

第一种使用形式如下。当指定的宏在程序中已定义，那么其中的语句序列参与源文件的编译，否则，其中的语句序列不参与源文件的编译。

```
`ifdef 宏名
```

语句序列

```
`endif
```

第二种使用形式如下。当指定的宏在程序中已定义，那么其中的语句序列1参与源文件的编译，否则，其中的语句序列2参与源文件的编译。

```
`ifdef 宏名
```

语句序列1

```
`else  
语句序列2  
`endif
```

2.6.6 行为语句的可综合性

前面几小节介绍了Verilog HDL中的多种行为语句，包括过程语句、赋值语句、条件语句、循环语句、编译指示语句，所有的语句都能在仿真中使用，但是有些语句是不可综合的，也就是说综合器无法将这些语句转变为对应的硬件电路。Verilog HDL行为语句可综合性的
情况如表2-4所示。

表2-4 Verilog HDL行为语句的可综合性

类别	语句	可综合性
过程语句	initial	
	always	√
赋值语句	持续赋值assign	√
	过程赋值=、<=	√
条件语句	if-else	√

	case	√
循环语句	for	√
	forever	
	repeat	
	while	
编译指示语句	`define	√
	`include	
	`ifdef、`else、`endif	√

2.7 电路设计举例

本节将设计一个简化的处理器取指令电路，通过这个例子体会 Verilog HDL的使用。

处理器内部一般有一个PC寄存器，其中存储指令地址，正常运行过程中，PC的值会随时间增加，同时从指令存储器中取出对应地址的指令。所以，本节实现的处理器取指令电路，包含两部分：PC模块、指令存储器。

1. PC模块的设计与实现

PC模块的功能就是给出取指令地址，同时每个时钟周期取指令地址递增。其接口设计如图2-13所示。采用左边是输入接口、右边是输

出接口的方式绘制，这样便于理解。接口作用描述如表2-5所示。

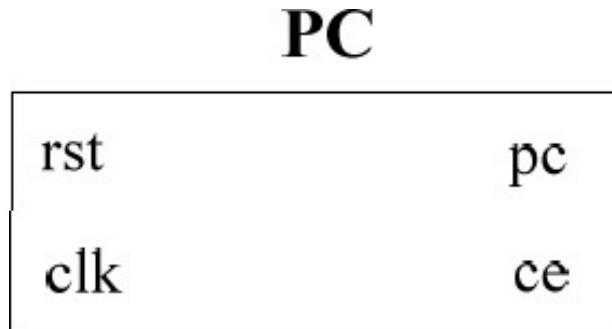


图2-13 PC模块的接口

表2-5 PC模块的接口描述

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	pc	6	输出	要读取的指令地址
4	ce	1	输出	指令存储器使能信号

此处定义指令地址pc的宽度为6，PC模块的主要代码如下，可以参考本书光盘Code\Chapter2目录下的pc_reg.v文件。

```
module pc_reg(  
  
    input wire      clk,  
    input wire      rst,  
  
    output reg[5:0]   pc,  
    output reg       ce  
  
);
```

```

always @ (posedge clk) begin //在时钟信号上升沿触发
    if (rst == 1'b1) begin
        ce <= 1'b0;           //复位信号有效的时候，指令存储器使能信号无效
    end else begin
        ce <= 1'b1;           //复位信号无效的时候，指令存储器使能信号有效
    end
end

always @ (posedge clk) begin //在时钟信号上升沿触发
    if (ce == 1'b0) begin
        pc <= 6'h00;         //指令存储器使能信号无效的时候，pc保持为0
    end else begin
        pc <= pc + 1'b1;     //指令存储器使能信号有效的时候，pc在每个时钟加1
    end
end

endmodule

```

2. 指令存储器ROM的设计与实现

指令存储器ROM的作用是存储指令，并依据输入的地址，给出对应地址的指令。其接口如图2-14所示，还是采用左边是输入接口、右边是输出接口的方式绘制，这样便于理解。接口描述如表2-6所示。

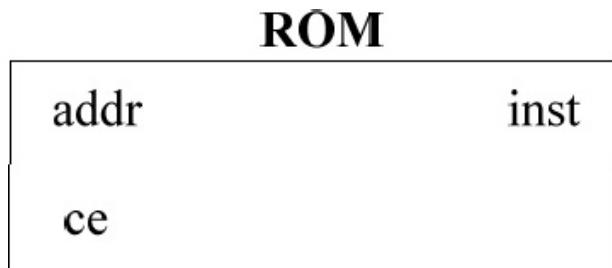


图2-14 ROM模块的接口

表2-6 ROM模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	ce	1	输入	使能信号
2	addr	6	输入	要读取的指令地址
3	inst	32	输出	读出的指令

此处定义指令的宽度为32，指令存储器ROM的主要代码如下，可以参考在本书光盘Code\Chapter2目录下的rom.v文件。

```
module rom(  
  
    input  wire          ce,  
    input  wire[5:0]     addr,  
    output reg[31:0]    inst  
  
);  
  
reg[31:0]  rom[63:0];           //使用二维向量定义存储器  
  
always @ (*) begin  
    if (ce == 1'b0) begin  
        inst <= 32'h0;           //使能信号无效时，给出的数据是0  
    end else begin  
        inst <= rom[addr];      //使能信号有效时，给出地址addr对应的指令  
    end  
end
```

```
endmodule
```

其中使用了一个二维向量定义存储器，深度是64，每个元素的宽度是32，这也是使用6位地址即可的原因。

3. 顶层文件

先介绍元件例化的知识，在一个复杂电路的实现过程中，可以将其分割成多个功能单元分别实现，然后在一个顶层文件中通过调用各个功能单元，将其按照一定方式连接在一起，从而实现最终电路。其中调用功能单元的过程就称为元件例化。元件例化的格式如图2-15所示。

```
<模块名> <实例名>(  
    .<相连的端口名>(相连的信号名),  
    .<相连的端口名>(相连的信号名),  
    .....  
)
```

图2-15 元件例化的格式

经过上面两步，我们分别实现了PC模块、指令存储器ROM，现在可以编写顶层文件将两者连接起来。连接方式如图2-16所示。

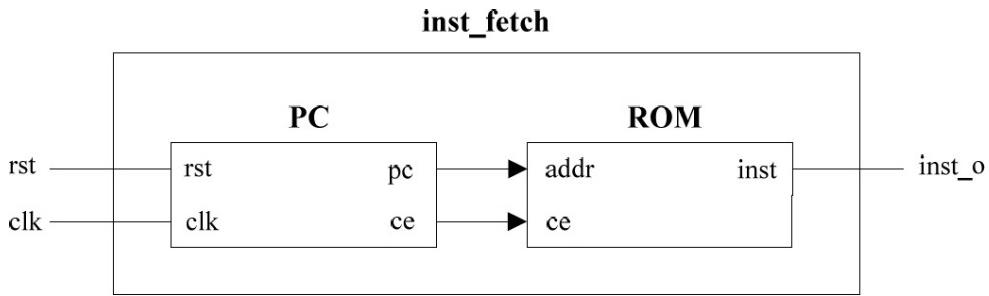


图2-16 顶层文件inst_fetch

PC模块的输出pc连接到指令存储器ROM的地址接口addr，PC模块输出的使能信号ce连接到ROM的使能信号接口ce。顶层模块对应的模块名为inst_fetch，有三个接口，接口描述如表2-7所示。

表2-7 顶层文件inst_fetch模块的接口描述

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	inst_o	32	输出	读出的指令

inst_fetch模块的主要代码如下，其中例化了PC模块、指令存储器ROM。可以参考本书光盘Code\Chapter2目录下的inst_fetch.v文件。

```

module inst_fetch(
    input wire    clk,
    input wire    rst,
    output wire[31:0]  inst_o
);

    wire[5:0] pc;

```

```
    wire      rom_ce;

    //PC模块的例化
    pc_reg pc_reg0(.clk(clk), .rst(rst),
                  .pc(pc), .ce(rom_ce));

    //指令存储器ROM的例化
    rom rom0( .ce(rom_ce), .addr(pc), .inst(inst_o));

endmodule
```

PC模块的输出pc、ROM模块的输入addr都连接到变量pc，所以两者连接在一起；PC模块的输出ce、ROM模块的输入ce都连接到rom_ce，所以两者连接在一起。这样就实现了图2-16所示的连接关系。

2.8 仿真

2.7节实现了一个简化的处理器取指电路，需要通过仿真以验证其功能是否正确，直观的仿真思路就是：给出一个时钟信号，上述电路会在每个时钟信号上升沿将取指地址加1，同时从指令存储器中取出一条指令，观察取指地址是否依次递增，同时观察取出的指令是否是存储器中取指地址对应的指令，如果都符合，那么上述取指电路就实现正确。此处涉及两个问题。

(1) 如何在指令存储器中存储指令，也就是指令存储器初始化问题。

(2) 如何给出时钟信号？

本节将分别解答上述问题，在此基础上，使用ModelSim进行仿真。

2.8.1 系统函数

初始化存储器有两种方法，一种是对存储器中每个存储单元依次给出初值，如下。

```
rom[0] = 32'h00000000;      //存储器rom的第0个元素初始化为0x00000000
rom[1] = 32'h01010101;      //存储器rom的第1个元素初始化为0x01010101
rom[2] = 32'h02020202;      //存储器rom的第2个元素初始化为0x02020202
rom[3] = 32'h03030303;      //存储器rom的第3个元素初始化为0x03030303
....
```

另一种方法是使用系统函数\$readmemh，这样更加方便，但是后者只能用于仿真中。

除了\$readmemh外，在Verilog HDL中还定义了很多系统函数，比如显示当前仿真时间的函数\$time、显示信号值的函数\$display、暂停仿真过程的函数\$stop、结束仿真过程的函数\$finish等。本书主要用到了\$stop、\$readmemh这两个系统函数。

1. \$stop

\$stop用于对仿真过程进行控制，暂停仿真，其使用格式如下。

```
$stop();           // 使用格式一，不带参数  
$stop(n);         // 使用格式二，带参数n，n可以等于0、1、2等值，含义如下：  
                  // 0：不输出任何信息；  
                  // 1：给出仿真时间和位置  
                  // 2：给出仿真时间和位置，还有其他一些运行统计数据
```

当仿真程序执行到\$stop语句时，将暂时停止仿真，此时设计者可以输入命令，对仿真器进行交互控制。

2. \$readmemh

\$readmemh函数用于读取文件，其作用是从外部文件中读取数据并放入存储器中。使用格式如下。

```
$readmemh("数据文件名", 存储对象);
```

将第1个参数指定文件的数据读入第2个参数指定的存储器中。例如。

```
reg[31:0] rom[63:0];  
  
initial $readmemh ("rom.data", rom); // 读入文件rom.data的数据  
到rom中
```

此处对数据文件的格式有一定要求，要求使用十六进制记录数据，且每一行记录一个地址的数据。例如：rom.data的内容如下，每
一行是一个32位的数据。

```
00000000  
01010101  
02020202  
03030303  
.....
```

使用\$readmemh ("rom.data", rom)函数后， rom的内容就会初始化为如下。 rom[0]: 32'h00000000; //存储器rom的第0个元素初始化为0x00000000

```
rom[1]: 32'h01010101; //存储器rom的第1个元素初始化为0x01010101  
rom[2]: 32'h02020202; //存储器rom的第2个元素初始化为0x02020202  
rom[3]: 32'h03030303; //存储器rom的第3个元素初始化为0x03030303  
.....
```

回到本节最开始提出的两个问题，现在可以回答第一个问题了，为了实现对指令存储器的初始化，只需要创建一个数据文件，其内容如上面的 rom.data 所示，然后在指令存储器 rom.v 中，增加代码 \$readmemh ("rom.data", rom)即可。完整代码可以参考本书光盘 Code\Chapter2 目录下的 rom.v 文件。

2.8.2 Test Bench

现在回答本节最开始提出的第二个问题，通过创建Test Bench文件以给出时钟信号。

Test Bench为测试或仿真一个Verilog HDL程序搭建了一个平台，我们给被测试的模块施加激励信号，通过观察被测试模块的输出响应，从而判断其逻辑功能实现得正确与否。如图2-17所示。

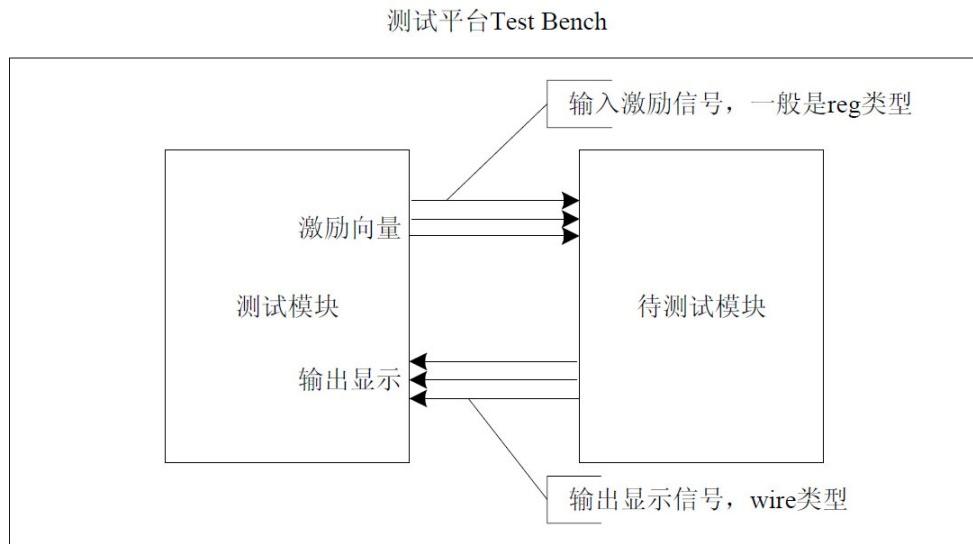


图2-17 Test Bench示意图

Test Bench的结构如图2-18所示，与2.4节介绍的Verilog HDL模块的结构没有根本区别，但有自身的一些特点。

- Test Bench只有模块名，没有端口列表；激励信号（输入到待测试模块的信号）必须定义为reg类型，以保持信号值；从待测试模块输出的信号（用户观察的信号）必须定义为wire类型。
- 在Test Bench中要调用被测试模块，也就是元件例化。
- Test Bench中一般会使用initial、always过程块来定义、描述激励信号。

```

module <Test Bench名>;
    <数据类型说明> // 激励信号使用reg类型，显示信号使用wire类型
    <激励向量定义> // always、initial过程块等
    <待测试模块例化>
endmodule

```

图2-18 Test Bench的一般结构

为简单取指令电路设计的Test Bench如下，完整代码位于本书光盘Code\Chapter2目录下的inst_fetch_tb.v文件。

```

module inst_fetch_tb;           // Test Bench名为inst_fetch_tb

/********************* 第一段：数据类型说明 *********************  

**  

*****  

***** 第一段： 数据类型说明  

*****  

*****  

**/  

    reg      CLOCK;          // 激励信号CLOCK，这是时钟信号  

    reg      rst;            // 激励信号rst，这是复位信号  

    wire[31:0] inst;         // 显示信号inst，取出的指令  

/********************* 第二段：激励向量定义 *********************  

**  

*****  

***** 第二段： 激励向量定义

```

```

*****
***** /



// 定义CLOCK信号，每隔10个时间单位，CLOCK的值翻转，由此得到一个周期信号。
// 在仿真的时候，一个时间单位默认是1ns，所以CLOCK的值每10ns翻转一次，对应

// 就是50MHz的时钟

initial begin
    CLOCK = 1'b0;
    forever #10 CLOCK = ~CLOCK;
end

// 定义rst信号，最开始为1，表示复位信号有效，过了195个时间单位，即195ns，

// 设置rst信号的值为0，复位信号无效，复位结束，再运行1000ns，暂停仿真
initial begin
    rst = 1'b1;
    #195 rst= 1'b0;
    #1000 $stop;
end

/*****



**



***** 第三段：待测试模块例化 *****



*****
```

```
/**/  
  
inst_fetch inst_fetch0(  
    .clk(CLOCK),  
    .rst(rst),  
    .inst_o(inst)  
)  
  
endmodule
```

2.8.3 ModelSim仿真

指令存储器初始化问题解决了、时钟信号也给出了，现在可以使用ModelSim进行仿真了。

1. 建立ModelSim工程

打开ModelSim，选择File->New->Project，出现新建工程对话框，其中填写工程名，选择保存目录，注意保存目录中不要有中文，如图2-19所示。

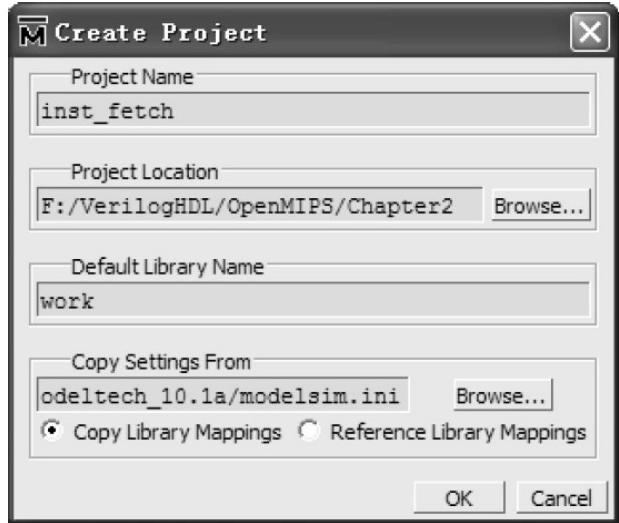


图2-19 新建ModelSim工程对话框

单击OK按钮后，会出现图2-20所示的界面，这里单击Add Existing File，也就是添加已有文件，出现图2-21所示的添加文件对话框。

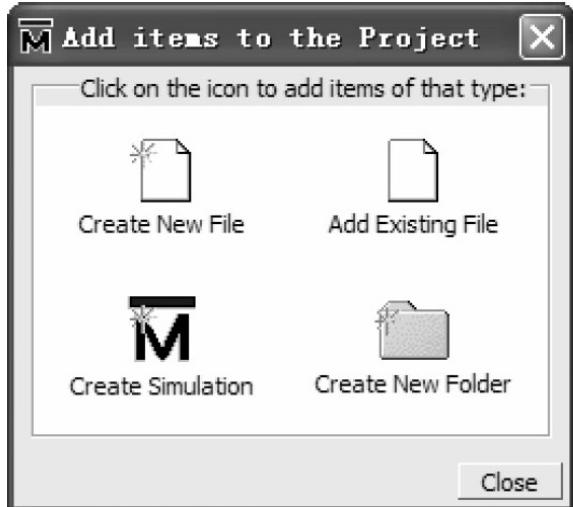


图2-20 选择添加已有文件

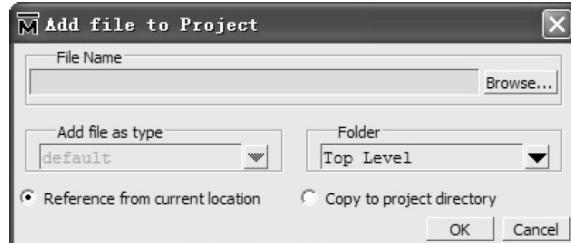


图2-21 添加文件对话框

单击Browse按钮，出现选择文件对话框，找到本书光盘的Code\Chapter2目录，添加其中所有的.v文件，如图2-22所示。

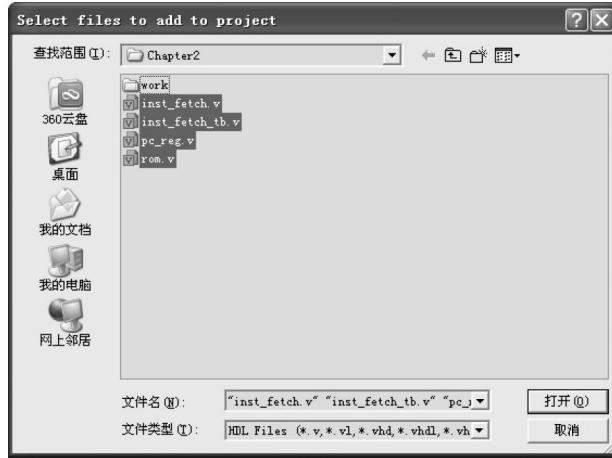


图2-22 添加Code\Chapter2目录下的所有.v文件

选择要添加的文件后，单击“打开”按钮，即完成添加，此时显示图2-23所示界面，在其中选中copy to project directory，这样就会将刚才选中的文件复制到新的工程目录下。

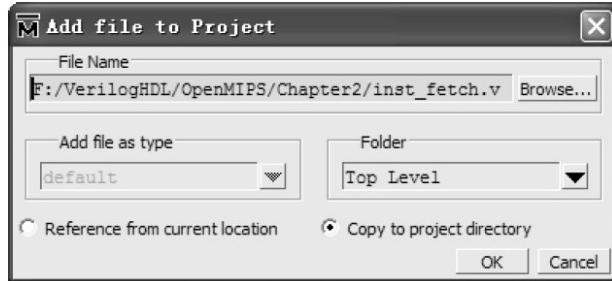


图2-23 选择Copy to project directory

文件添加完成后，会在ModelSim的主界面中显示所有文件的状态，其中问号表示对应文件没有编译。任意选中一个文件，用鼠标右键单击，在弹出菜单中选择Compile->Compile All，即开始编译所有文件，如图2-24所示。稍等几秒钟就编译结束了。编译结束后，所有的文件状态都应该是一个绿色的“√”。

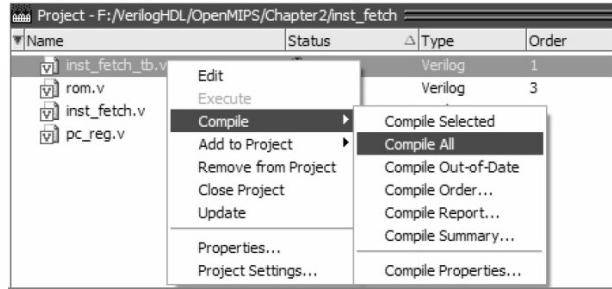


图2-24 编译所有文件

2. 开始仿真

切换到Library这个Tab，然后展开work目录，在inst_fetch_tb文件上单击右键，在弹出菜单中选择Simulate，如图2-25所示。

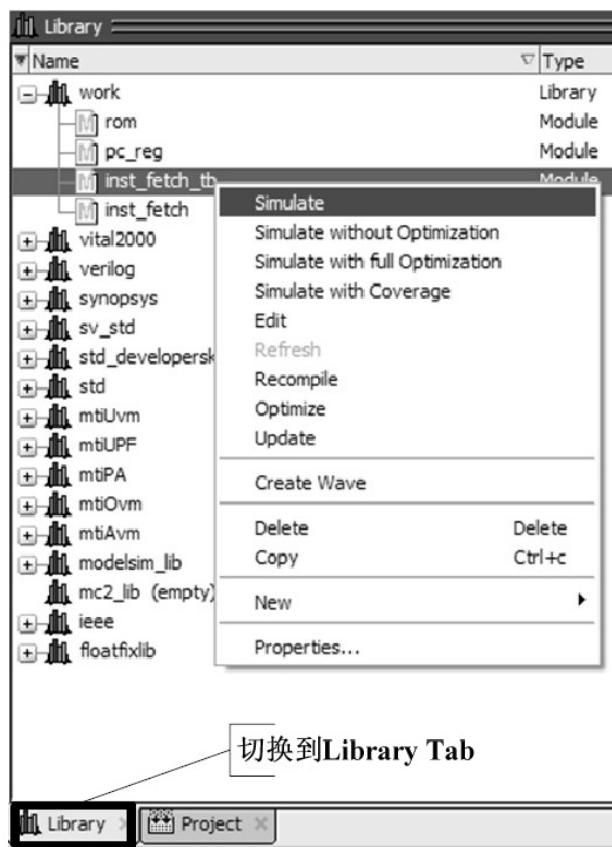


图2-25 在inst_fetch_tb上单击右键，选择Simulate

此时会增加一个Tab，名称为sim，展开其中的inst_fetch_tb节点，选择inst_fetch0，会在Objects窗口中显示inst_fetch模块的所有信号，如图2-26所示，如果没有出现Objects窗口，可以通过菜单View->Objects调出该窗口。

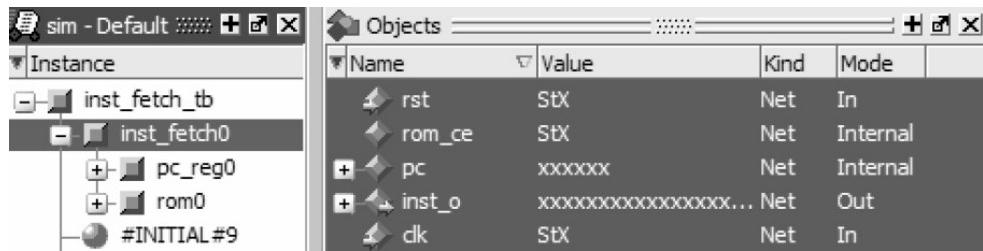


图2-26 在Objects窗口中显示选中模块的所有信号

选择Objects窗口中的所有信号，然后单击右键，在弹出菜单中选择Add to ->Wave->Selected Signals，如图2-27所示，将所有信号都添加到Wave窗口中。这些都是要观察的信号。添加要观察信号后的Wave窗口如图2-28所示。

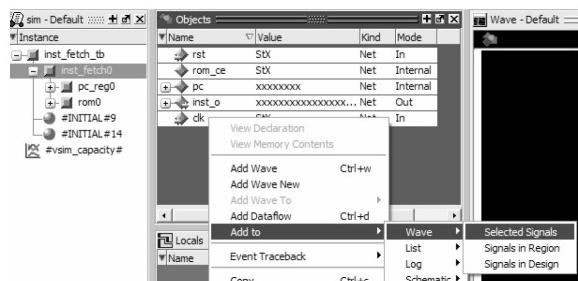


图2-27 选择要观察的信号

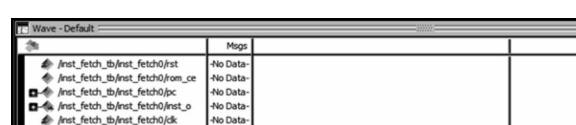


图2-28 添加要观察信号后的Wave窗口

单击工具栏中的Run-All按钮，就可开始仿真，如图2-29所示，仿真结果如图2-30所示。从仿真结果可知，处理器取指令电路实现正确。



图2-29 单击Run-All按钮开始仿真

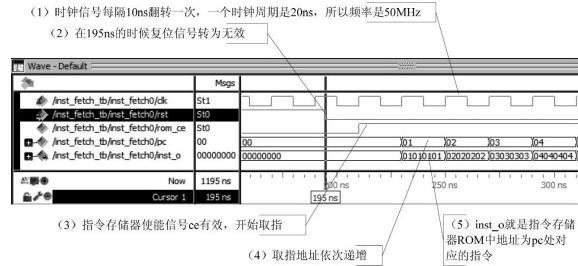


图2-30 仿真结果

2.9 本章小结

本章花了比较大的篇幅介绍了可编程逻辑器件的基本知识，以及基于可编程逻辑器件的数字系统设计流程，包括设计输入、综合、布局布线、下载、仿真等几步，这与传统的数字系统设计流程还是有很大不同的。然后介绍了Verilog HDL这样一种硬件编程语言，这也是将要用来实现OpenMIPS处理器的语言。在此基础上，设计实现了一个简化的处理器取指令电路，并使用ModelSim仿真验证该电路实现的正确性。在后期教学版OpenMIPS的设计实现过程中，主要也是使用ModelSim仿真验证，步骤都是一样的。

从第3章开始，就正式进入OpenMIPS处理器的设计实现阶段了。

第二篇 基础篇

第3章 教学版OpenMIPS处理器蓝图

第4章 第一条指令ori的实现

第5章 逻辑、移位操作与空指令的实现

第6章 移动操作指令的实现

第7章 算术操作指令的实现

第8章 转移指令的实现

第9章 加载存储指令的实现

第10章 协处理器访问指令的实现

第11章 异常相关指令的实现

第3章 教学版OpenMIPS处理器 蓝图

从本章开始将一步一步地实现教学版OpenMIPS处理器。本章给出了教学版OpenMIPS的系统蓝图，首先介绍了系统的设计目标，其中详细说明了OpenMIPS处理器计划实现的5级流水线。3.2节给出了OpenMIPS处理器的接口示意图，以及各个接口的作用。3.3节简单解释了各个源代码文件的作用，最后描述了OpenMIPS处理器的实现方法。读者将发现本书给出的实现方法与现有书籍的方法完全不同，更加易于理解、便于实践。

3.1 系统设计目标

3.1.1 设计目标

本书第二篇设计实现的教学版OpenMIPS处理器，是一款具有哈佛结构的32位标量处理器，兼容MIPS32 Release 1指令集架构（后文不再注明Release 1），这样的好处是可以使用现有的MIPS编译环境，如：GCC编译器等。OpenMIPS的设计目标如下。

- 五级整数流水线，分别是：取指、译码、执行、访存、回写。
- 哈佛结构，分开的指令、数据接口。
- 32个32位整数寄存器。
- 大端模式。

- 向量化异常处理，支持精确异常处理。
- 支持6个外部中断。
- 具有32bit数据、地址总线宽度。
- 能实现单周期乘法。
- 支持延迟转移。
- 兼容MIPS32指令集架构，支持MIPS32指令集中的所有整数指令。
- 大多数指令可以在一个时钟周期内完成。

上述设计目标都很容易理解，除了延迟转移和精确异常，前者将在第8章“转移指令的实现”中介绍，后者将在第11章“异常相关指令的实现”中介绍。

3.1.2 五级流水线

本书讲的是计算机中的流水线，首先看一下维基百科中对计算机流水线的定义：流水线是指将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快指令执行速度。此处有两个关键词：（1）拆分；（2）并行。指令的处理从直观上分析至少可以拆分为三步：从存储器取出指令、解释指令、按照解释的结果执行，简单地说就是：取指、译码、执行。如果我们只有一个硬件处理单元，这个单元既要取指，又要译码，还要执行，假设上述三种操作都可以在时间T完成，那么一条指令的处理时间为3T，n条指令的处理时间就为3nT，但是如果我们设计有三个硬件单元，分别做这三项工作的一项，那么就可以在执行的同时对下一条指令译码，在对下一条指

令译码的同时还可以再取一条指令，这就是经典的三级流水线，如图3-1所示。



图3-1 三级流水线示意图

从图3-1可知，在三级流水线上执行3条指令所需时间为 $5T$ ，而如果没有使用流水线则需要 $9T$ ，流水线确实加快了指令执行。ARM7采用的就是三级流水线。但世间是没有这么简单完美的，上面假设取指、译码、执行需要的时间都是 T ，实际并非如此，比如取指的时间就可能很长，假设取指需要 $2T$ 时间，那么如图3-2所示。

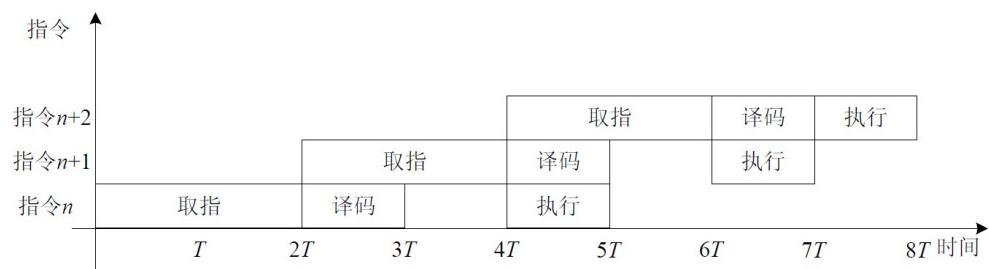


图3-2 取指时间为 $2T$ 时的流水线工作情况

可见在 $3T\sim 4T$ 的时间段、 $5T\sim 6T$ 的时间段，流水线在等待取指结束，此时译码阶段、执行阶段都停滞，这样一来自然就慢下来，最后，执行3条指令所需时间为 $8T$ 。解决取指时间过长的措施是引入缓存（Cache），处理器从缓存读取指令只需要1个时钟周期。

还有一种情况是执行阶段时间过长，比如指令为加载/存储指令（Load/Store）时，由于涉及访问存储器，执行阶段所需的时间就可能大于T，此时也会导致流水线停滞。为了解决这种情况下的流水线停滞问题，引入了五级流水线，分别是：取指、译码、执行、访存、回写。如图3-3所示。

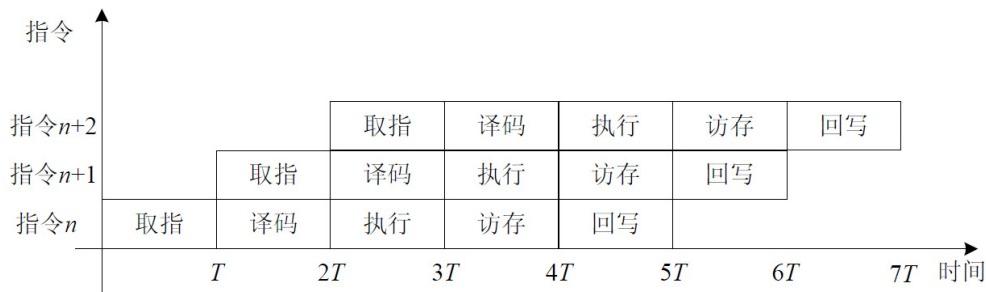


图3-3 五级流水线示意图

其中访存阶段（Memory Access）的作用是从存储器装载数据到寄存器或者将寄存器数据保存到存储器，当然，如果不是Load/Store指令则不需要这一步，此时在访存阶段就只是将执行阶段的运算结果送到下一级回写阶段。回写阶段（Write Back）的作用是将数据写入目的寄存器。ARM9就采用了这种五级流水线，OpenMIPS的设计目标也是五级流水线。具体而言，OpenMIPS五级流水线各个阶段的主要工作如下。

- **取指阶段：**从指令存储器读出指令，同时确定下一条指令地址。
- **译码阶段：**对指令进行译码，从通用寄存器中读出要使用的寄存器的值，如果指令中含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令，并且满足转移条件，那么给出转移目标，作为新的指令地址。

- **执行阶段：**按照译码阶段给出的操作数、运算类型，进行运算，给出运算结果。如果是Load/Store指令，那么还会计算Load/Store的目标地址。
- **访存阶段：**如果是Load/Store指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时，在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址处继续执行。
- **回写阶段：**将运算结果保存到目标寄存器。

读者可能对上述流水线各个阶段的主要工作还不完全理解，没关系，本书也不是一次实现上述全部工作，而是逐步完善，一开始，只实现流水线各个阶段的基本工作，慢慢地丰富、完善。

3.1.3 指令执行周期

OpenMIPS设计目标中提到：实现MIPS32指令集中的所有整数指令，并且大多数指令可以在一个时钟周期内执行完成。具体而言，OpenMIPS实现的所有指令执行完成需要的时钟周期如表3-1所示。

表3-1 OpenMIPS中所有指令执行完成需要的时钟周期

指令	执行完成需要的时钟周期
除法指令div、divu	36
乘累加指令madd、maddu	2

乘累减指令msub、msubu	2
其余指令	1

对表3-1有以下几点说明。

(1) OpenMIPS计划采用试商法完成除法运算，对于32位的除法，执行阶段至少需要32个时钟周期，再加上一些准备工作需要的时钟周期，最后需要36个时钟周期才能执行完成。在第7章“算术操作指令的实现”中会具体介绍除法指令的实现过程。

(2) 乘累加指令madd、maddu，乘累减指令msub、msubu都需要2个时钟周期才能执行完成。主要是因为这4条指令都要做两次运算，一次乘法、一次加/减法，如果将这两次运算放在执行阶段的一个时钟周期中完成，那么会使执行阶段所需要的时间明显增加，从而降低OpenMIPS工作时钟的频率，因此，OpenMIPS设计在执行阶段使用两个时钟周期完成这4条指令，一个时钟周期进行乘法，下一个时钟周期进行加/减法。在第7章“算术操作指令的实现”中会具体介绍乘累加、乘累减指令的实现过程。

3.2 教学版OpenMIPS处理器接口

教学版OpenMIPS处理器的外部接口如图3-4所示。采用左边是输入接口，右边是输出接口的方式绘制，这样比较直观，便于理解。各接口的描述如表3-2所示，可以分为三类：系统控制接口（包括复位、时钟、中断）、指令存储器接口、数据存储器接口。

OpenMIPS (教学版)

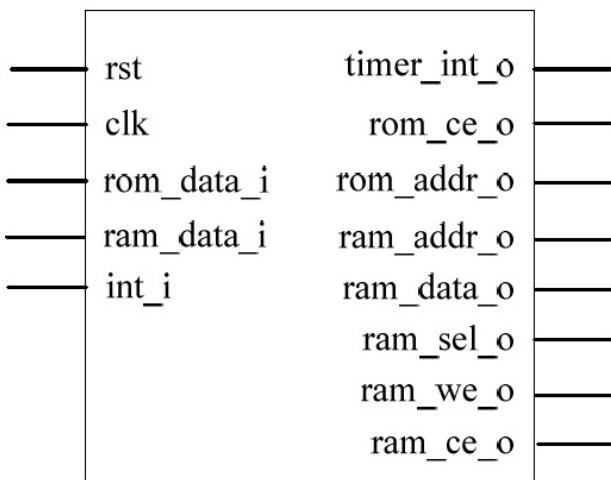


图3-4 教学版OpenMIPS处理器的外部接口

表3-2 教学版OpenMIPS处理器外部接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	rom_data_i	32	输入	从指令存储器取得的指令
4	rom_addr_o	32	输出	输出到指令存储器的地址
5	rom_ce_o	1	输出	指令存储器使能信号
6	ram_data_i	32	输入	从数据存储器读取的数据
7	ram_addr_o	32	输出	要访问的数据存储器的地址
8	ram_we_o	1	输出	是否是对数据存储器的写操作, 为1表示是写操作

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
9	ram_sel_o	4	输出	字节选择信号
10	ram_data_o	32	输出	要写入数据存储器的数据
11	ram_ce_o	1	输出	数据存储器使能信号
12	int_i	6	输入	6个外部硬件中断输入
13	timer_int_o	1	输出	定时器中断信号

3.3 文件说明

OpenMIPS是五级流水线处理器，流水线各个阶段的模块、对应的文件如图3-5所示。图中每个模块的上方标注的是模块名，下方标注的

是对应的文件名。模块之间的关系没有绘出，因为关系比较复杂，在书中不便绘制，读者可以参考本书光盘中的“openmips模块连接关系图.vsd”文件，其中绘制了模块之间详细的连接关系。

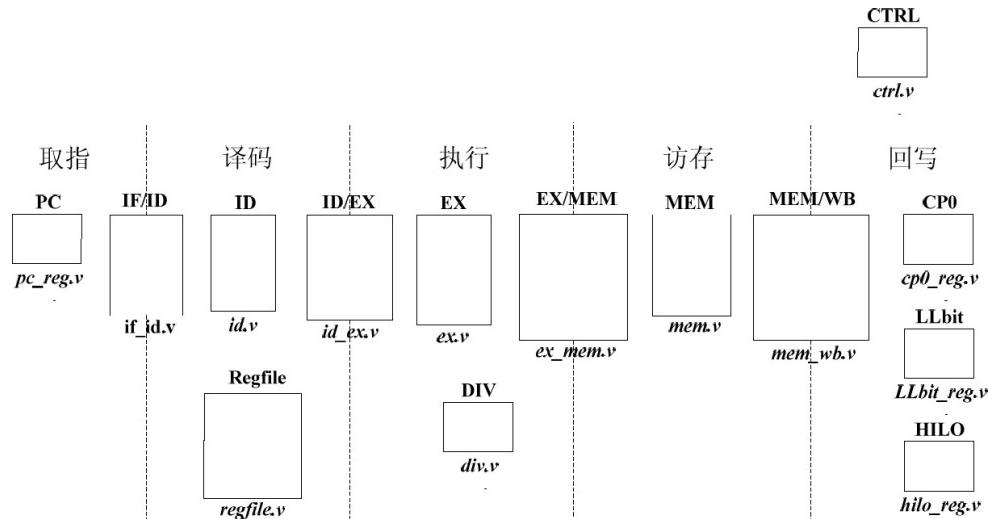


图3-5 OpenMIPS流水线各个阶段的模块、对应的文件

图3-5具体说明如下。

(1) 取指阶段

- PC模块：给出指令地址，其中实现指令指针寄存器PC，该寄存器的值就是指令地址，对应`pc_reg.v`文件。
- IF/ID模块：实现取指与译码阶段之间的寄存器，将取指阶段的结果（取得的指令、指令地址等信息）在下一个时钟传递到译码阶段，对应`if_id.v`文件。

(2) 译码阶段

- ID模块：对指令进行译码，译码结果包括运算类型、运算所需的源操作数、要写入的目的寄存器地址等，对应`id.v`文件。

- Regfile模块：实现了32个32位通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作，对应regfile.v文件。
- ID/EX模块：实现译码与执行阶段之间的寄存器，将译码阶段的结果在下一个时钟周期传递到执行阶段，对应id_ex.v文件。

(3) 执行阶段

- EX模块：依据译码阶段的结果，进行指定的运算，给出运算结果。对应ex.v文件。
- DIV模块：进行除法运算的模块，对应div.v文件。
- EX/MEM模块：实现执行与访存阶段之间的寄存器，将执行阶段的结果在下一个时钟周期传递到访存阶段，对应ex_mem.v文件。

(4) 访存阶段

- MEM模块：如果是加载、存储指令，那么会对数据存储器进行访问。此外，还会在该模块进行异常判断。对应mem.v文件。
- MEM/WB模块：实现访存与回写阶段之间的寄存器，将访存阶段的结果在下一个时钟周期传递到回写阶段，对应mem_wb.v文件。

(5) 回写阶段

- CP0模块：对应MIPS架构中的协处理器CP0。

- LLbit模块：实现寄存器LLbit，在链接加载指令ll、条件存储指令sc的处理过程中会使用到该寄存器，第9章会详述。
- HILO模块：实现寄存器HI、LO，在乘法、除法指令的处理过程中会使用到这两个寄存器，第7章会详述。

另外，还有一个CTRL模块，在图3-5的上部单独画出。这个模块对应ctrl.v文件，是用来控制整个流水线的暂停、清除等动作，所以不便于将其归入流水线中的某一个阶段。

本书的附录A给出了各个模块的接口示意图，以及接口的作用描述。读者可能会感觉模块太多、每个模块的接口也太多，似乎很难理解，这种担心是不必要的，之前也提到本书不是一次实现上述全部模块，而是首先实现其中一部分模块，这一部分模块也只实现少量接口，只要能满足我们的要求即可，然后随着OpenMIPS实现的指令种类越来越多，慢慢添加模块、增加接口。

3.4 实现方法

在写作本书之前，已经出现了一些介绍软核处理器实现的书籍，这些书在介绍实现方法时有一个共同点：一次考虑所有的指令、所有的情况，然后给出代码。笔者认为这种方法读者不易于接受，而且这也不是作者实现处理器时采用的方法。在本书中，笔者借鉴了软件开发中的“增量模型”概念，使用了一种完全不同的实现方法：先考虑最简单的情况，给出代码，然后考虑稍微多一点的情况，修改、补充代码，随着考虑情况的增多，不停地修改、补充代码，最终，使代码实现需求；同时，也符合读者的认识过程。

在第4章中，我们就考虑了一种最简单的情况——只实现一条指令，这条指令是逻辑“或”指令ori，借助这条指令，可以搭建OpenMIPS流水线的结构，此时的数据流图如图3-6所示。读者暂时不用理解其具体含义，只需与图3-7对比，体会各自的复杂度，具体含义会在第4章介绍。

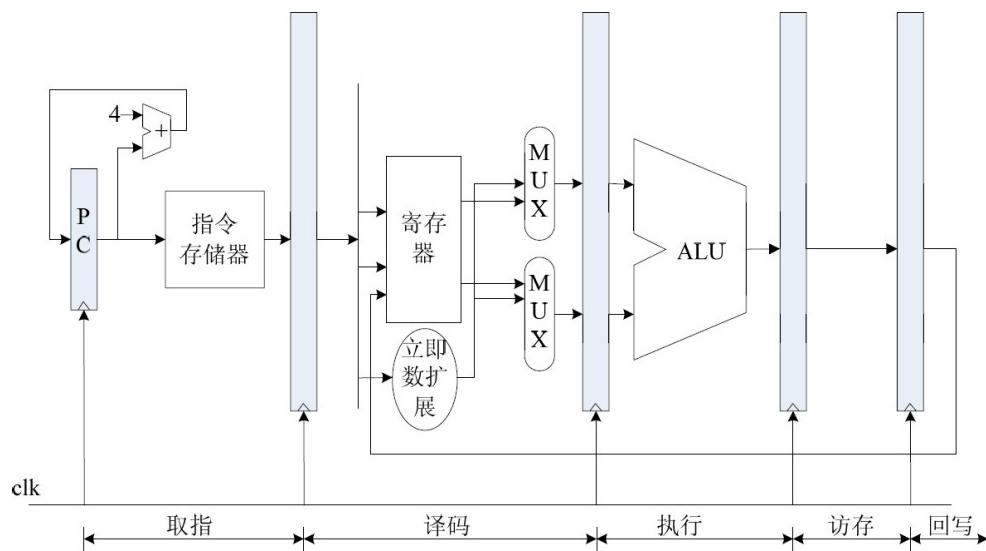


图3-6 只实现一条指令ori时的数据流图

在后续章节中我们依次实现逻辑操作指令、移位操作指令、空指令、移动操作指令、算术操作指令、转移指令、加载存储指令、协处理器访问指令、异常相关指令，最终实现MIPS32指令集架构中定义的所有整数指令，此时的数据流图如图3-7所示。同样，读者此时暂不用理解其具体含义，只需与图3-6对比，体会各自的复杂度。

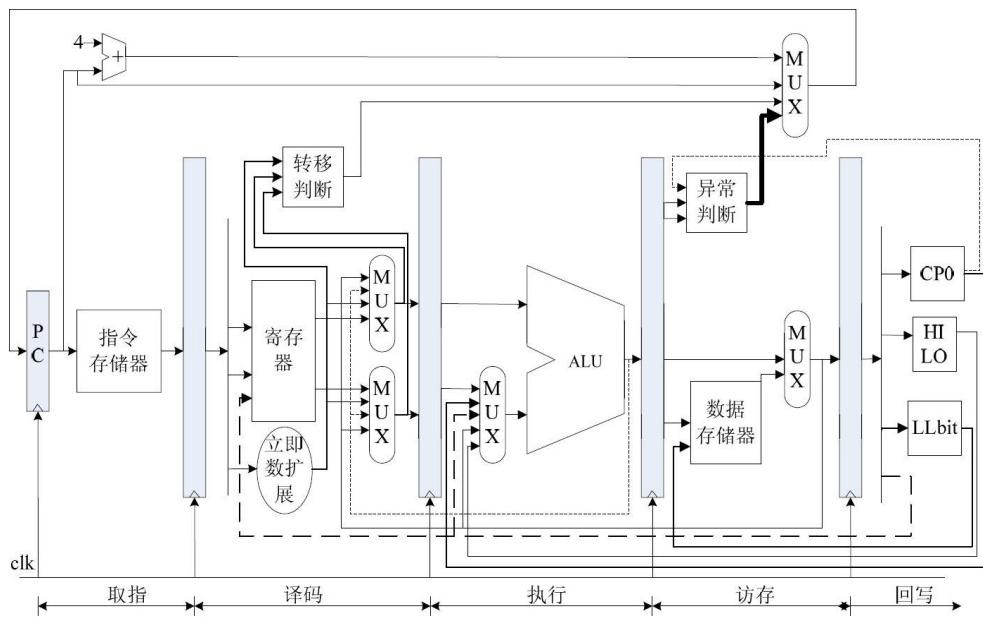


图3-7 实现MIPS32指令集中所有整数指令之后的数据流图

对比图3-6、图3-7，会发现复杂度大大增加，如果笔者一开始就考虑实现图3-7所示的数据流图，那么读者需要知道MIPS32指令集中定义的所有指令，还要理解其作用。显然会增加理解难度。更好的方法是，一开始只考虑图3-6所示的数据流图，读者只需要理解指令ori的作用，然后一步步添加实现更多指令，同时丰富完善数据流图，最终实现图3-7所示的数据流图。比如：在添加实现转移指令的时候，就会为图3-6数据流图中的译码阶段增加“转移判断”模块；在添加实现异常相关指令的时候，就会为图3-6数据流图中的访存阶段增加“异常判断”模块。

以上就是本书在实现OpenMIPS处理器时采用的实现方法，第4章将实现最小结构，即只考虑执行一条指令ori。

第4章 第一条指令ori的实现

前面几章介绍了很多预备知识，也描绘了即将要实现的OpenMIPS处理器的蓝图，各位读者是不是早已摩拳擦掌，迫切希望一展身手了，好吧，本章我们将实现OpenMIPS处理器的第一条指令ori，为什么选择这条指令作为我们实现的第一条指令呢？答案就两个字——简单，指令ori用来实现逻辑“或”运算，选择一条简单的指令有助于我们排除干扰，将注意力集中在流水线结构的实现上，当然也可以选择其他类似的指令，只要简单即可。通过这条简单指令的实现，本章在4.2节将初步建立OpenMIPS的五级流水线结构，当我们在后面章节中实现其余指令的时候，都是在这个初步建立的流水线结构上进行扩充。

在ori指令实现后，要验证其实现是否正确，所以在4.3节建立了最小SOPC，仅仅包含OpenMIPS、指令存储器，用于验证ori指令是否实现正确，后续章节验证其余指令的时候，都是在这个最小SOPC或者其改进模型上进行验证的。

本章最后介绍了MIPS编译环境的建立。

4.1 ori指令说明

ori是进行逻辑“或”运算的指令，其指令格式如图4-1所示。

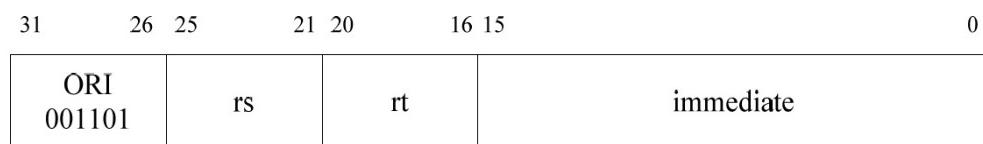


图4-1 ori指令格式

从指令格式中可以知道，这是一个I类型的指令，ori指令的指令码是6'b001101，所以当处理器发现正在处理的指令的高6bit是6'b001101时，就知道当前正在处理的是ori指令。

指令用法为：ori rs, rt, immediate，作用是将指令中的16位立即数immediate进行无符号扩展至32位，然后与索引为rs的通用寄存器的值进行逻辑“或”运算，运算结果保存到索引为rt的通用寄存器中。这里需要说明以下两点。

(1) 无符号扩展

在MIPS32指令集架构中，经常会有指令需要将其中的立即数进行符号扩展，或者无符号扩展，一般都是将n位立即数扩展为32位，其中，符号扩展是将n位立即数的最高位复制到扩展后的32位数据的高（32-n）位，无符号扩展则是将扩展后的32位数据的高（32-n）位都置为0。以将指令中的16位立即数扩展为32位为例，表4-1给出了当16位立即数分别是0x8000、0x1000时的符号扩展、无符号扩展的结果。

表4-1 16位立即数扩展举例

16位立即数	0x8000	0x1000
符号扩展	0xFFFF8000	0x00001000
无符号扩展	0x00008000	0x00001000

(2) 通用寄存器

在MIPS32指令集架构中定义了32个通用寄存器\$0-\$31，OpenMIPS实现了这32个通用寄存器，使用某一个通用寄存器只需要给出相应索引，这

一个索引占用5bit，ori指令中的rs、rt就是通用寄存器的索引，例如：当rs为5'b00011时，就表示通用寄存器\$3。

4.2 流水线结构的建立

4.2.1 流水线的简单模型

数字电路有组合逻辑、时序逻辑之分，其中时序逻辑最基本的器件是寄存器，此处的寄存器不是在4.1节中提到的MIPS架构规定的通用寄存器\$0-\$31，后者是一个更高层面的概念，前者是类似于D触发器这种数字电路的基本器件。寄存器按照给定时间脉冲来进行时序同步操作，其使得时序逻辑电路具有记忆功能。而组合逻辑电路则由逻辑门组成，提供电路的所有逻辑功能。实际的数字电路一般是组合逻辑与时序逻辑的结合。如果寄存器的输出端和输入端存在环路，这样的电路称为“状态机”。状态机的简单模型如图4-2所示。如果寄存器之间有连接，而没有上述环路，这样的电路结构称为“流水线”。流水线的简单模型如图4-3所示。

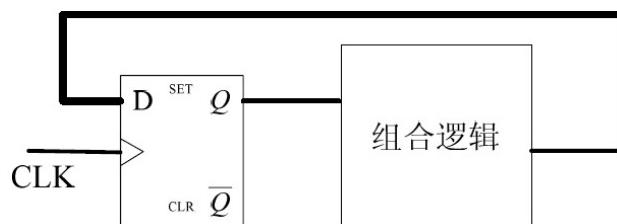


图4-2 状态机的简单模型



图4-3 流水线的简单模型

在流水线结构中，信号在寄存器之间传递，每传递到一级都会引起相应的组合逻辑电路变化，对这种模型进行抽象描述就是寄存器传输级（Register Transfer Level，RTL）。本节接下来要实现的原始的OpenMIPS五级流水线结构就是图4-3的扩充。

4.2.2 原始的OpenMIPS五级流水线结构

扩充图4-3，可以得到OpenMIPS的原始数据流图，如图4-4所示。这个数据流图还很不完整，在后续章节中会随着实现指令的增加而丰富，但这个原始的数据流图已经可以表达本节要实现的ori指令在流水线中的处理过程了。

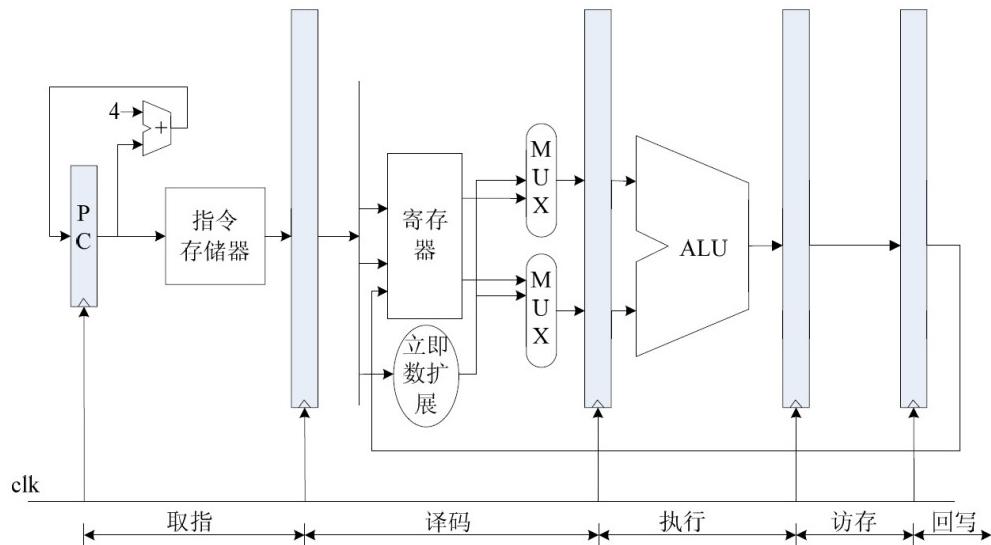


图4-4 原始的数据流图

图中深色部分对应的是图4-3中的D触发器，深色部分之间的部分对应的是图4-3中的组合逻辑。各个阶段完成的主要工作如下。

- 取指：取出指令存储器中的指令，PC值递增，准备取下一条指令。
- 译码：对指令进行译码，依据译码结果，从32个通用寄存器中取出源操作数，有的指令要求两个源操作数都是寄存器的值，比如or指令，有的指令要求其中一个源操作数是指令中立即数的扩展，比如ori指令，所以这里有两个复用器，用于依据指令要求，确定参与运算的操作数，最终确定的两个操作数会送到执行阶段。
- 执行阶段：依据译码阶段送入的源操作数、操作码，进行运算，对于ori指令而言，就是进行逻辑“或”运算，运算结果传递到访存阶段。
- 访存阶段：对于ori指令，在访存阶段没有任何操作，直接将运算结果向下传递到回写阶段。
- 回写阶段：将运算结果保存到目的寄存器。

图4-5是为实现上述数据流图而设计的OpenMIPS五级流水线系统结构图，图中显示了各个模块的接口、连接关系。每个模块上方是模块名，下方是对应的Verilog HDL程序文件名。本章的4.2.4~4.2.8节将分别介绍图中各个模块的实现。

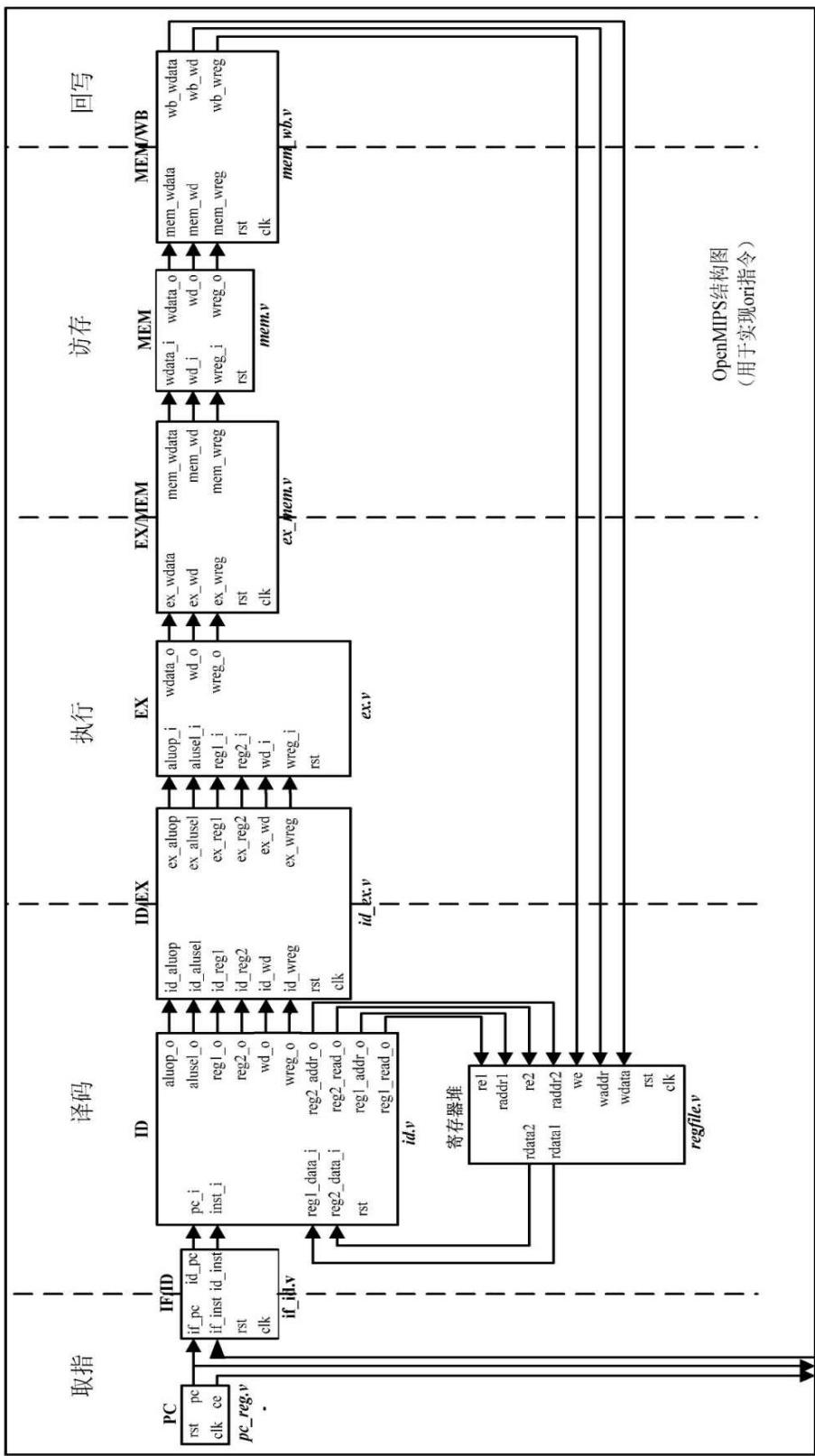


图4-5 原始的OpenMIPS五级流水线系统结构图

指令存储器

OpenMIPS结构图
(用于实现ori指令)

4.2.3 一些宏定义

在正式开始介绍流水线结构实现之前，需要给出一些宏定义，因为在OpenMIPS的实现过程中，为了提高代码的可读性和易懂性，使用了较多的宏，全部的宏都在文件defines.v中定义。此处列举在本章中会使用的一部分宏，后面随着OpenMIPS功能的不断完善，会有更多的宏添加进来，届时会对新增加的宏进行说明。

```
//*****
***** 全局的宏定义
*****  
`define RstEnable           1'b1           //复位信号有效  
`define RstDisable          1'b0           //复位信号无效  
`define ZeroWord            32'h00000000 //32位的数值0  
`define WriteEnable         1'b1           //使能写  
`define WriteDisable        1'b0           //禁止写  
`define ReadEnable          1'b1           //使能读  
`define ReadDisable         1'b0           //禁止读  
`define AluOpBus            7:0           //译码阶段的输出  
aluop_o的宽度  
`define AluSelBus           2:0           //译码阶段的输出  
alusel_o的宽度  
`define InstValid           1'b0           //指令有效  
`define InstInvalid          1'b1           //指令无效  
`define True_v               1'b1           //逻辑“真”  
`define False_v              1'b0           //逻辑“假”  
`define ChipEnable           1'b1           //芯片使能  
`define ChipDisable          1'b0           //芯片禁止
```

```
//*****与具体指令有关的宏定义*****
`define EXE_ORI           6'b001101          //指令ori的指令码
`define EXE_NOP            6'b0000000

//AluOp
`define EXE_OR_OP          8'b00100101
`define EXE_NOP_OP          8'b000000000

//AluSel
`define EXE_RES_LOGIC      3'b001
`define EXE_RES_NOP         3'b000

//*****与指令存储器ROM有关的宏定义*****
`define InstAddrBus        31:0             //ROM的地址总线宽度
`define InstBus              31:0             //ROM的数据总线宽度
`define InstMemNum          131071           //ROM的实际大小为128KB
`define InstMemNumLog2       17                //ROM实际使用的地址线宽度

//*****与通用寄存器Regfile有关的宏定义*****
```

```

*****
`define RegAddrBus          4:0                      //Regfile模块的地址
线宽度
`define RegBus               31:0                     //Regfile模块的数据
线宽度
`define RegWidth             32                       //通用寄存器的宽度
`define DoubleRegWidth       64                       //两倍的通用寄存器的
宽度
`define DoubleRegBus         63:0                     //两倍的通用寄存器的
数据线宽度
`define RegNum               32                       //通用寄存器的数量
`define RegNumLog2           5                        //寻址通用寄存器使用的
地址位数
`define NOPRegAddr           5'b00000

```

4.2.4 取指阶段的实现

取指阶段取出指令存储器中的指令，同时，PC值递增，准备取下一条指令，包括PC、IF/ID两个模块。

1. PC模块

PC模块的作用是给出指令地址，其接口描述如表4-2所示。

表4-2 PC模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	pc	32	输出	要读取的指令地址
4	ce	1	输出	指令存储器使能信号

PC模块对应的源文件是pc_reg.v，代码如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。读者可以使用任何文本编辑工具编辑该文件，笔者习惯使用UltraEdit，所有的代码都是使用它编辑的，当然也可以使用Windows自带的记事本。

```
module pc_reg(
    input wire          clk,
    input wire          rst,
    output reg[`InstAddrBus] pc,
    output reg          ce
);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        ce <= `ChipDisable;      // 复位的时候指令存储器禁用
    end else begin
        ce <= `ChipEnable;     // 复位结束后，指令存储器使能
    end
end

always @ (posedge clk) begin
    if (ce == `ChipDisable) begin
        pc <= 32'h00000000;   // 指令存储器禁用的时候，PC为0
    end else begin
        pc <= pc + 4'h4;      // 指令存储器使能的时候，PC的值每时钟周期加4
    end
end
```

```
endmodule
```

其中使用到了一些define.v中定义的宏，InstAddrBus宏表示指令地址线的宽度，此处定义为32，RstEnable宏表示复位信号有效，定义为1'b1，也就是当输入rst为高电平时，表示复位信号有效。

在复位的时候，输出的指令存储器使能信号为ChipDisable，表示指令存储器禁用，其余时刻指令存储器使能信号为ChipEnable，表示指令存储器使能。

当指令存储器禁用时，PC的值保持为0，当指令存储器能使用时，PC的值会在每时钟周期加4，表示下一条指令的地址，因为一条指令是32位，而我们设计的OpenMIPS是可以按照字节寻址的，一条指令对应4个字节，所以PC加4指向下一条指令地址。读者需要注意区分：在2.7节设计的简单取指电路是按照字寻址的，所以每时钟周期PC加1。

2. IF/ID模块

IF/ID模块的作用是暂时保存取指阶段取得的指令，以及对应的指令地址，并在下一个时钟传递到译码阶段，其接口描述如表4-3所示。

表4-3 IF/ID模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	if_pc	32	输入	取指阶段取得的指令对应的地址
4	if_inst	32	输入	取指阶段取得的指令
5	id_pc	32	输出	译码阶段的指令对应的地址
6	id_inst	32	输出	译码阶段的指令

IF/ID模块对应的源代码文件是if_id.v，代码如下，读者可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module if_id(
    input wire clk,
    input wire rst,
    //来自取指阶段的信号，其中宏定义InstBus表示指令宽度，为32
    input wire[`InstAddrBus]      if_pc,
    input wire[`InstBus]          if_inst,
    //对应译码阶段的信号
    output reg[`InstAddrBus]     id_pc,
    output reg[`InstBus]          id_inst
);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        id_pc <= `ZeroWord;      // 复位的时候pc为0
        id_inst <= `ZeroWord;    // 复位的时候指令也为0，实际就是空
   指令
        end else begin
            id_pc <= if_pc;       // 其余时刻向下传递取指阶段的值
            id_inst <= if_inst;
        end
    end
endmodule
```

从代码可以知道，其中只有一个时序电路，IF/ID模块只是简单地将取指阶段的结果在每个时钟周期的上升沿传递到译码阶段。

4.2.5 译码阶段的实现

参考图4-5可知，IF/ID模块的输出连接到ID模块，那么，我们的指令此时已经进入译码阶段，在此阶段，将对取到的指令进行译码：给出要进行的运算类型，以及参与运算的操作数。译码阶段包括Regfile、ID和ID/EX三个模块。

1. Regfile模块

Regfile模块实现了32个32位通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作，其接口描述如表4-4所示。

表4-4 Regfile模块接口描述表

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据

续表

序号	接口名	宽度(bit)	输入/输出	作用
5	we	1	输入	写使能信号
6	raddr1	5	输入	第一个读寄存器端口要读取的寄存器的地址
7	re1	1	输入	第一个读寄存器端口读使能信号
8	rdata1	32	输出	第一个读寄存器端口输出的寄存器值
9	raddr2	5	输入	第二个读寄存器端口要读取的寄存器的地址
10	re2	1	输入	第二个读寄存器端口读使能信号
11	rdata2	32	输出	第二个读寄存器端口输出的寄存器值

Regfile模块对应的源代码文件是regfile.v，代码如下，可以在本书附带光盘的Code\Chapter4\目录下找到regfile.v文件。

```

module regfile(
    input wire clk,
    input wire rst,

    // 写端口
    input wire           we,
    input wire[`RegAddrBus]   waddr,
    input wire[`RegBus]      wdata,

    // 读端口1
    input wire           re1,
    input wire[`RegAddrBus]   raddr1,
    output reg[`RegBus]     rdata1,

    // 读端口2
    input wire           re2,
    input wire[`RegAddrBus]   raddr2,
    output reg[`RegBus]     rdata2
);

//***** 第一段：定义32个32位寄存器 *****
***** 第一段：定义32个32位寄存器 *****

reg[`RegBus]  regs[0:`RegNum-1];

//***** 第二段：写操作 *****
***** 第二段：写操作 *****
```

```

***** */

always @ (posedge clk) begin
    if (rst == `RstDisable) begin
        if((we == `WriteEnable) && (waddr != `RegNumLog2'h0))
begin
            regs[waddr] <= wdata;
        end
    end
end

/*****
第三段：读端口1的读操作
*****/

```

```

always @ (*) begin
    if(rst == `RstEnable) begin
        rdata1 <= `ZeroWord;
    end else if(raddr1 == `RegNumLog2'h0) begin
        rdata1 <= `ZeroWord;
    end else if((raddr1 == waddr) && (we == `WriteEnable)
                && (re1 == `ReadEnable)) begin
        rdata1 <= wdata;
    end else if(re1 == `ReadEnable) begin
        rdata1 <= regs[raddr1];
    end else begin
        rdata1 <= `ZeroWord;
    end
end

```

```

/*
*****第四段：读端口2的读操作*****
*/
always @ (*) begin
    if(rst == `RstEnable) begin
        rdata2 <= `ZeroWord;
    end else if(raddr2 == `RegNumLog2'h0) begin
        rdata2 <= `ZeroWord;
    end else if((raddr2 == waddr) && (we == `WriteEnable)
                && (re2 == `ReadEnable)) begin
        rdata2 <= wdata;
    end else if(re2 == `ReadEnable) begin
        rdata2 <= regs[raddr2];
    end else begin
        rdata2 <= `ZeroWord;
    end
end

endmodule

```

Regfile模块可以分为四段进行理解。

(1) 第一段：定义了一个二维的向量，元素个数是RegNum，这是在defines.v中的一个宏定义，为32，每个元素的宽度是RegBus，这也是在defines.v中的一个宏定义，也为32，所以此处定义的就是32个32位寄存器。

(2) 第二段：实现了写寄存器操作，当复位信号无效时（rst为RstDisable），在写使能信号we有效（we为WriteEnable），且写操作目的寄存器不等于0的情况下，可以将写输入数据保存到目的寄存器。之所以要判断目的寄存器不为0，是因为MIPS32架构规定\$0的值只能为0，所以不要写入。WriteEnable是defines.v中定义的宏，表示写使能信号有效，这些宏定义的含义十分明显，从名称上就可以知道具体含义，所以本书后面对宏定义不再作出说明，除非这个宏定义的含义从名称上不易明白。

(3) 第三段：实现了第一个读寄存器端口，分以下几步依次判断：

- 当复位信号有效时，第一个读寄存器端口的输出始终为0；
- 当复位信号无效时，如果读取的是\$0，那么直接给出0；
- 如果第一个读寄存器端口要读取的目标寄存器与要写入的目的寄存器是同一个寄存器，那么直接将要写入的值作为第一个读寄存器端口的输出；
- 如果上述情况都不满足，那么给出第一个读寄存器端口要读取的目标寄存器地址对应寄存器的值；
- 第一个读寄存器端口不能使用时，直接输出0。

(4) 第四段：实现了第二个读寄存器端口，具体过程与第三段是相似的，不再重复解释。

注意一点：读寄存器操作是组合逻辑电路，也就是一旦输入的要读取的寄存器地址raddr1或者raddr2发生变化，那么会立即给出新地址对应的寄存器的值，这样可以保证在译码阶段取得要读取的寄存器的值，而写寄存器操作是时序逻辑电路，写操作发生在时钟信号的上升沿。

2. ID模块

ID模块的作用是对指令进行译码，得到最终运算的类型、子类型、源操作数1、源操作数2、要写入的目的寄存器地址等信息，其中运算类型指的是逻辑运算、移位运算、算术运算等，子类型指的是更加详细的运算类型，比如：当运算类型是逻辑运算时，运算子类型可以是逻辑“或”运算、逻辑“与”运算、逻辑“异或”运算等。ID模块的接口描述如表4-5所示。

表4-5 ID模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	pc_i	32	输入	译码阶段的指令对应的地址
3	inst_i	32	输入	译码阶段的指令

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
4	reg1_data_i	32	输入	从 Regfile 输入的第一个读寄存器端口的输入
5	reg2_data_i	32	输入	从 Regfile 输入的第二个读寄存器端口的输入
6	reg1_read_o	1	输出	Regfile 模块的第一个读寄存器端口的读使能信号
7	reg2_read_o	1	输出	Regfile 模块的第二个读寄存器端口的读使能信号
8	reg1_addr_o	5	输出	Regfile 模块的第一个读寄存器端口的读地址信号
9	reg2_addr_o	5	输出	Regfile 模块的第二个读寄存器端口的读地址信号
10	aluop_o	8	输出	译码阶段的指令要进行的运算的子类型
11	alusel_o	3	输出	译码阶段的指令要进行的运算的类型
12	reg1_o	32	输出	译码阶段的指令要进行的运算的源操作数 1
13	reg2_o	32	输出	译码阶段的指令要进行的运算的源操作数 2
14	wd_o	5	输出	译码阶段的指令要写入的目的寄存器地址
15	wreg_o	1	输出	译码阶段的指令是否有要写入的目的寄存器

ID模块对应的代码文件是id.v，其内容如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module id(
    input wire      rst,
    input wire[`InstAddrBus] pc_i,
    input wire[`InstBus]          inst_i,
    // 读取的Regfile的值
```

```

    input wire[`RegBus]           reg1_data_i,
    input wire[`RegBus]           reg2_data_i,

    // 输出到Regfile的信息
    output reg                   reg1_read_o,
    output reg                   reg2_read_o,
output reg[`RegAddrBus]       reg1_addr_o,
    output reg[`RegAddrBus]       reg2_addr_o,

    // 送到执行阶段的信息
    output reg[`AluOpBus]        aluop_o,
    output reg[`AluSelBus]        alusel_o,
    output reg[`RegBus]          reg1_o,
    output reg[`RegBus]          reg2_o,
    output reg[`RegAddrBus]       wd_o,
    output reg                  wreg_o
);

// 取得指令的指令码，功能码
// 对于ori指令只需通过判断第26-31bit的值，即可判断是否是ori指令
wire[5:0] op   = inst_i[31:26];
wire[4:0] op2 = inst_i[10:6];
wire[5:0] op3 = inst_i[5:0];
wire[4:0] op4 = inst_i[20:16];

// 保存指令执行需要的立即数
reg[`RegBus] imm;

```

```

// 指示指令是否有效

reg instvalid;

/*
***** 第一段：对指令进行译码 *****
*/
always @ (*) begin
    if (rst == `RstEnable) begin
        aluop_o      <= `EXE_NOP_OP;
        alusel_o     <= `EXE_RES_NOP;
        wd_o         <= `NOPRegAddr;
        wreg_o       <= `WriteDisable;
        instvalid   <= `InstValid;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= `NOPRegAddr;
        reg2_addr_o <= `NOPRegAddr;
        imm          <= 32'h0;
    end else begin
        aluop_o      <= `EXE_NOP_OP;
        alusel_o     <= `EXE_RES_NOP;
        wd_o         <= inst_i[15:11];
        wreg_o       <= `WriteDisable;
        instvalid   <= `InstInvalid;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= inst_i[25:21]; // 默认通过Regfile读端口1读取的
    end
end

```

寄存器地址

```
reg2_addr_o <= inst_i[20:16]; // 默认通过Regfile读端口2读取的  
寄存器地址
```

```
imm <= `ZeroWord;
```

case (op)

```
`EXE_ORI: begin // 依据op的值判断是否是ori指令
```

```
// ori指令需要将结果写入目的寄存器，所以wreg_o为WriteEnable  
wreg_o <= `WriteEnable;
```

```
// 运算的子类型是逻辑“或”运算
```

```
aluop_o <= `EXE_OR_OP;
```

```
// 运算类型是逻辑运算
```

```
alusel_o <= `EXE_RES_LOGIC;
```

```
// 需要通过Regfile的读端口1读取寄存器
```

```

    reg1_read_o <= 1'b1;

    // 不需要通过Regfile的读端口2读取寄存器
    reg2_read_o <= 1'b0;

    // 指令执行需要的立即数
    imm          <= {16'h0, inst_i[15:0]};

    // 指令执行要写的目的寄存器地址
    wd_o         <= inst_i[20:16];

    // ori指令是有效指令
    instvalid   <= `InstValid;

end

default: begin
end

endcase //case op

end      //if

end      //always

//*****************************************************************************
***** 第二段：确定进行运算的源操作数1 *****
//*************************************************************************/

```

```

always @ (*) begin
    if(rst == `RstEnable) begin
        reg1_o <= `ZeroWord;

```

```
end else if(reg1_read_o == 1'b1) begin
    reg1_o <= reg1_data_i;      // Regfile读端口1的输出值
end else if(reg1_read_o == 1'b0) begin
    reg1_o <= imm;           // 立即数
end else begin
    reg1_o <= `ZeroWord;
end
end
```

```
/*************************************************************************
***** 第三段：确定进行运算的源操作数2 *****
*/
```

```
always @ (*) begin
    if(rst == `RstEnable) begin
        reg2_o <= `ZeroWord;
    end else if(reg2_read_o == 1'b1) begin
        reg2_o <= reg2_data_i;      // Regfile读端口2的输出值
    end else if(reg2_read_o == 1'b0) begin
        reg2_o <= imm;           // 立即数
    end else begin
        reg2_o <= `ZeroWord;
    end
end
```

```
endmodule
```

ID模块中的电路都是组合逻辑电路，另外，从图4-5可知ID模块与Regfile模块也有接口连接。其代码可以分为三段进行理解。

(1) 第一段：实现了对指令的译码，依据指令中的特征字段区分指令，对指令ori而言，只需通过识别26-31bit的指令码是否为6'b001101，即可判断是否是ori指令，其中的宏定义EXE_ORI就是6'b001101，op就是指令的26-31bit，所以当op等于EXE_ORI时，就表示是ori指令，此时会有以下译码结果。

- 要读取的寄存器情况：ori指令只需要读取rs寄存器的值，默认通过Regfile读端口1读取的寄存器地址reg1_addr_o的值是指令的21-25bit，参考图4-1可知，正是ori指令中的rs，所以设置reg1_read_o为1，通过图4-5可以看出，reg1_read_o连接Regfile的输入re1，reg1_addr_o连接Regfile的输入raddr1，结合对Regfile模块的介绍可知，译码阶段会读取寄存器rs的值。指令ori需要的另一个操作数是立即数，所以设置reg2_read_o为0，表示不通过Regfile读端口2读取寄存器，这里暗含使用立即数作为运算的操作数。imm就是指令中的立即数进行零扩展后的值。
- 要执行的运算：alusel_o给出要执行的运算类型，对于ori指令而言就是逻辑操作，即EXE_RES_LOGIC。aluop_o给出要执行的运算子类型，对于ori指令而言就是逻辑“或”运算，即EXE_OR_OP。这两个值会传递到执行阶段。
- 要写入的目的寄存器：wreg_o表示是否要写目的寄存器，ori指令要将计算结果保存到寄存器中，所以wreg_o设置为WriteEnable。wd_o是要写入的目的寄存器地址，此时就是指令的16-20bit，参考图4-1可知，正是ori指令中的rt。这两个值也会传递到执行阶段。

(2) 第二段：给出参与运算的源操作数1的值，如果reg1_read_o为1，那么就将从Regfile模块读端口1读取的寄存器的值作为源操作数1，如果reg1_read_o为0，那么就将立即数作为源操作数1，对于ori而言，此处选择从Regfile模块读端口1读取的寄存器的值作为源操作数1。该值将通过reg1_o端口被传递到执行阶段。

(3) 第三段：给出参与运算的源操作数2的值，如果reg2_read_o为1，那么就将从Regfile模块读端口2读取的寄存器的值作为源操作数2，如果reg2_read_o为0，那么就将立即数作为源操作数2，对于ori而言，此处选择立即数imm作为源操作数2。该值将通过reg2_o端口被传递到执行阶段。

3. ID/EX模块

参考图4-5可知，ID模块的输出连接到ID/EX模块，后者的作用是将译码阶段取得的运算类型、源操作数、要写的目的寄存器地址等结果，在下一个时钟传递到流水线执行阶段。其接口描述如表4-6所示。

表4-6 ID/EX模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	id_alusel	3	输入	译码阶段的指令要进行的运算的类型
4	id_aluop	8	输入	译码阶段的指令要进行的运算的子类型
5	id_reg1	32	输入	译码阶段的指令要进行的运算的源操作数1
6	id_reg2	32	输入	译码阶段的指令要进行的运算的源操作数2
7	id_wd	5	输入	译码阶段的指令要写入的目的寄存器地址
8	id_wreg	1	输入	译码阶段的指令是否有要写入的目的寄存器
9	ex_alusel	3	输出	执行阶段的指令要进行的运算的类型
10	ex_aluop	8	输出	执行阶段的指令要进行的运算的子类型
11	ex_reg1	32	输出	执行阶段的指令要进行的运算的源操作数1
12	ex_reg2	32	输出	执行阶段的指令要进行的运算的源操作数2
13	ex_wd	5	输出	执行阶段的指令要写入的目的寄存器地址
14	ex_wreg	1	输出	执行阶段的指令是否有要写入的目的寄存器

ID/EX模块对应的代码文件是id_ex.v，其内容如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module id_ex(  
  
    input wire      clk,  
    input  wire      rst,  
  
    // 从译码阶段传递过来的信息  
    input wire[`AluOpBus]      id_aluop,  
    input wire[`AluSelBus]     id_alusel,  
    input wire[`RegBus]       id_reg1,  
    input wire[`RegBus]       id_reg2,  
    input wire[`RegAddrBus]   id_wd,  
    input wire              id_wreg,  
  
    // 传递到执行阶段的信息  
    output reg[`AluOpBus]     ex_aluop,  
    output reg[`AluSelBus]    ex_alusel,  
    output reg[`RegBus]       ex_reg1,  
    output reg[`RegBus]       ex_reg2,  
    output reg[`RegAddrBus]   ex_wd,  
    output reg                ex_wreg  
  
);  
  
    always @ (posedge clk) begin
```

```

if (rst == `RstEnable) begin
    ex_aluop  <= `EXE_NOP_OP;
    ex_alusel <= `EXE_RES_NOP;
    ex_reg1   <= `ZeroWord;
    ex_reg2   <= `ZeroWord;
    ex_wd     <= `NOPRegAddr;
    ex_wreg   <= `WriteDisable;
end else begin
    ex_aluop  <= id_aluop;
    ex_alusel <= id_alusel;
    ex_reg1   <= id_reg1;
    ex_reg2   <= id_reg2;
    ex_wd     <= id_wd;
    ex_wreg   <= id_wreg;
end
end

endmodule

```

代码十分清晰，其中只有一个时序电路，ID/EX模块只是简单地将译码阶段的结果在时钟周期的上升沿传递到执行阶段。执行阶段将依据这些值进行运算。

4.2.6 执行阶段的实现

现在，指令已经进入流水线的执行阶段了，在此阶段将依据译码阶段的结果，对源操作数1、源操作数2，进行指定的运算。执行阶段包括EX、EX/MEM两个模块。

1. EX模块

观察图4-5中ID/EX与EX模块的端口连接关系可知，EX模块会从ID/EX模块得到运算类型alusel_i、运算子类型aluop_i、源操作数reg1_i、源操作数reg2_i、要写的目的寄存器地址wd_i。EX模块会依据这些数据进行运算，其接口描述如表4-7所示。

表4-7 EX模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	alusel_i	3	输入	执行阶段要进行的运算的类型
3	aluop_i	8	输入	执行阶段要进行的运算的子类型
4	reg1_i	32	输入	参与运算的源操作数1
5	reg2_i	32	输入	参与运算的源操作数2
6	wd_i	5	输入	指令执行要写入的目的寄存器地址
7	wreg_i	1	输入	是否有要写入的目的寄存器
8	wd_o	5	输出	执行阶段的指令最终要写入的目的寄存器地址
9	wreg_o	1	输出	执行阶段的指令最终是否有要写入的目的寄存器
10	wdata_o	32	输出	执行阶段的指令最终要写入目的寄存器的值

EX模块对应的代码文件为ex.v，其内容如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module ex(  
  
    input wire          rst,  
  
    // 译码阶段送到执行阶段的信息  
    input wire[`AluOpBus]      aluop_i,  
    input wire[`AluSelBus]     alusel_i,  
    input wire[`RegBus]        reg1_i,  
    input wire[`RegBus]        reg2_i,  
    input wire[`RegAddrBus]    wd_i,
```

```
    input wire                      wreg_i,  
  
    // 执行的结果  
    output reg[`RegAddrBus]         wd_o,  
    output reg                      wreg_o,  
    output reg[`RegBus]            wdata_o  
  
);  
  
    // 保存逻辑运算的结果  
    reg[`RegBus] logicout;  
  
/****** 第一段：依据aluop_i指示的运算子类型进行运算，此处只有逻辑“或”运算 ***/  
/****** */  
/  
  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        logicout <= `ZeroWord;  
    end else begin  
        case (aluop_i)  
  
`EXE_OR_OP: begin
```

```

logicout <= reg1_i | reg2_i;

end

default: begin
    logicout <= `ZeroWord;
end
endcase
end      //if
end      //always

//*****************************************************************************
** 第二段：依据alusel_i指示的运算类型，选择一个运算结果作为最终结果 **
** 此处只有逻辑运算结果
***** */

always @ (*) begin
    wd_o    <= wd_i;           // wd_o等于wd_i，要写的目的寄存器地址
    wreg_o <= wreg_i;         // wreg_o等于wreg_i，表示是否要写目的
    寄存器

```

```

        case ( alusel_i )
            `EXE_RES_LOGIC: begin
                wdata_o <= logicout; // wdata_o中存放运算结果
            end
            default: begin
                wdata_o <= `ZeroWord;
            end
        endcase
    end

endmodule

```

EX模块中都是组合逻辑电路，上述代码可以分为两段理解。

(1) 第一段：依据输入的运算子类型进行运算，这里只有一种，就是逻辑“或”运算，运算结果保存在logicout中，这个变量专门用来保存逻辑操作的结果，以后还会添加算术运算、移位运算等，届时，会定义一些新的变量保存对应的运算结果。

(2) 第二段：给出最终的运算结果，包括是否要写目的寄存器wreg_o、要写的目的寄存器地址wd_o、要写入的数据wdata_o。其中wreg_o、wd_o的值都直接来自译码阶段，不需要改变，wdata_o的值要依据运算类型进行选择，如果是逻辑运算，那么将logicout的值赋给wdata_o。此处实际上是为以后扩展做准备，当添加其他类型的指令时，只需要修改这里的case情况即可。

2. EX/MEM模块

参考图4-5可知，EX模块的输出连接到EX/MEM模块，后者的作用是将执行阶段取得的运算结果，在下一个时钟传递到流水线访存阶段，其接

口描述如表4-8所示。

表4-8 EX/MEM模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	ex_wd	5	输入	执行阶段的指令执行后要写入的目的寄存器地址
4	ex_wreg	1	输入	执行阶段的指令执行后是否有要写入的目的寄存器
5	ex_wdata	32	输入	执行阶段的指令执行后要写入目的寄存器的值
6	mem_wd	5	输出	访存阶段的指令要写入的目的寄存器地址
7	mem_wreg	1	输出	访存阶段的指令是否有要写入的目的寄存器
8	mem_wdata	32	输出	访存阶段的指令要写入目的寄存器的值

EX/MEM模块对应的代码文件是ex_mem.v，内容如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module ex_mem(  
  
    input wire      clk,  
    input  wire       rst,  
  
    // 来自执行阶段的信息  
    input wire[`RegAddrBus]      ex_wd,  
    input  wire           ex_wreg,  
    input wire[`RegBus]        ex_wdata,  
  
    // 送到访存阶段的信息  
    output reg[`RegAddrBus]      mem_wd,  
    output  reg           mem_wreg,  
    output reg[`RegBus]        mem_wdata  
);
```

```

always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        mem_wd      <= `NOPRegAddr;
        mem_wreg    <= `WriteDisable;
        mem_wdata   <= `ZeroWord;
    end else begin
        mem_wd      <= ex_wd;
        mem_wreg    <= ex_wreg;
        mem_wdata   <= ex_wdata;
    end
end

endmodule

```

十分简单，其中只有一个时序逻辑电路，在时钟上升沿，将执行阶段的结果传递到访存阶段。

4.2.7 访存阶段的实现

现在，ori指令进入访存阶段了，但是由于ori指令不需要访问数据存储器，所以在访存阶段，不做任何事，只是简单地将执行阶段的结果向回写阶段传递即可。

流水线访存阶段包括MEM、MEM/WB两个模块。

1. MEM模块

MEM模块的接口描述如表4-9所示。

表4-9 MEM模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	wd_i	5	输入	访存阶段的指令要写入的目的寄存器地址
3	wreg_i	1	输入	访存阶段的指令是否有要写入的目的寄存器
4	wdata_i	32	输入	访存阶段的指令要写入目的寄存器的值
5	wd_o	5	输出	访存阶段的指令最终要写入的目的寄存器地址
6	wreg_o	1	输出	访存阶段的指令最终是否有要写入的目的寄存器
7	wdata_o	32	输出	访存阶段的指令最终要写入目的寄存器的值

MEM模块的代码位于文件mem.v，内容如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module mem(  
  
    input wire      rst,  
  
    // 来自执行阶段的信息  
    input wire[`RegAddrBus]      wd_i,  
    input wire                  wreg_i,  
    input wire[`RegBus]         wdata_i,  
  
    // 访存阶段的结果  
    output reg[`RegAddrBus]      wd_o,  
    output reg                  wreg_o,  
    output reg[`RegBus]         wdata_o  
);  
  
    always @ (*) begin  
        if(rst == `RstEnable) begin
```

```

        wd_o      <= `NOPRegAddr;
        wreg_o   <= `WriteDisable;
        wdata_o <= `ZeroWord;
    end else begin
        wd_o      <= wd_i;
        wreg_o   <= wreg_i;
        wdata_o <= wdata_i;
    end
end

endmodule

```

MEM模块中只有一个组合逻辑电路，将输入的执行阶段的结果直接作为输出，参考图4-5可知，MEM模块的输出连接到MEM/WB模块。

2. MEM/WB模块

MEM/WB模块的作用是将访存阶段的运算结果，在下一个时钟传递到回写阶段，其接口描述如表4-10所示。

表4-10 MEM/WB模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	mem_wd	5	输入	访存阶段的指令最终要写入的目的寄存器地址
4	mem_wreg	1	输入	访存阶段的指令最终是否有要写入的目的寄存器
5	mem_wdata	32	输入	访存阶段的指令最终要写入目的寄存器的值
6	wb_wd	5	输出	回写阶段的指令要写入的目的寄存器地址

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
7	wb_wreg	1	输出	回写阶段的指令是否有要写入的目的寄存器
8	wb_wdata	32	输出	回写阶段的指令要写入目的寄存器的值

MEM/WB模块的代码位于mem_wb.v文件，其主要内容如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module mem_wb(  
  
    input wire      clk,  
    input wire      rst,  
  
    // 访存阶段的结果  
    input wire[`RegAddrBus]      mem_wd,  
    input wire                  mem_wreg,  
    input wire[`RegBus]         mem_wdata,  
  
    // 送到回写阶段的信息  
    output reg[`RegAddrBus]     wb_wd,  
    output reg                  wb_wreg,  
    output reg[`RegBus]        wb_wdata  
  
);  
  
    always @ (posedge clk) begin  
        if(rst == `RstEnable) begin  
            wb_wd      <= `NOPRegAddr;  
            wb_wreg   <= `WriteDisable;  
            wb_wdata  <= `ZeroWord;  
        end else begin  
            wb_wd      <= mem_wd;  
            wb_wreg   <= mem_wreg;
```

```
    wb_wdata <= mem_wdata;  
  end  
end  
  
endmodule
```

MEM/WB的代码与MEM模块的代码十分相似，都是将输入信号传递到对应的输出端口，但是MEM/WB模块中的是时序逻辑电路，即在时钟上升沿才发生信号传递，而MEM模块中的是组合逻辑电路。MEM/WB模块将访存阶段指令是否要写目的寄存器mem_wreg、要写的目的寄存器地址mem_wd、要写入的数据mem_wdata等信息传递到回写阶段对应的接口wb_wreg、wb_wd、wb_wdata。

4.2.8 回写阶段的实现

经过上面的传递，ori指令的运算结果已经进入回写阶段了，这个阶段实际上是在Regfile模块中实现的，从图4-5可知，MEM/WB模块的输出wb_wreg、wb_wd、wb_wdata连接到Regfile模块，分别连接到写使能端口we、写操作目的寄存器端口waddr、写入数据端口wdata，所以会将指令的运算结果写入目的寄存器，具体代码可以参考Regfile模块。

4.2.9 顶层模块OpenMIPS的实现

顶层模块OpenMIPS在文件openmips.v中实现，主要内容就是对上面实现的流水线各个阶段的模块进行例化、连接，连接关系就如图4-5所示。在本章实现的OpenMIPS模块的接口如图4-6所示，还是采用左边是输入接口，右边是输出接口的方式绘制，便于理解，各接口的说明如表4-11所

示。可见与第3章的系统蓝图还有较大差距，很多接口都没有，在后续章节随着OpenMIPS实现指令的增多，会逐步完善，最终实现第3章的系统蓝图。

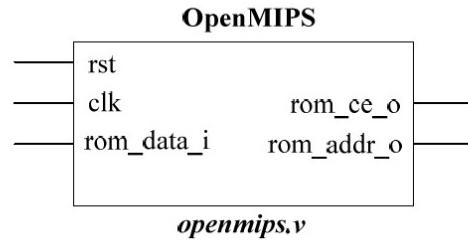


图4-6 OpenMIPS模块接口图

表4-11 OpenMIPS模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	rom_data_i	32	输入	从指令存储器取得的指令
4	rom_addr_o	32	输出	输出到指令存储器的地址
5	rom_ce_o	1	输出	指令存储器使能信号

代码如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module openmips(  
  
    input wire          clk,  
    input wire          rst,  
  
    input wire[`RegBus]      rom_data_i,  
    output wire[`RegBus]     rom_addr_o,  
    output wire           rom_ce_o  
);
```

```
// 连接IF/ID模块与译码阶段ID模块的变量  
wire[`InstAddrBus] pc;  
wire[`InstAddrBus] id_pc_i;  
wire[`InstBus] id_inst_i;  
  
// 连接译码阶段ID模块输出与ID/EX模块的输入的变量  
wire[`AluOpBus] id_aluop_o;  
wire[`AluSelBus] id_alusel_o;  
wire[`RegBus] id_reg1_o;  
wire[`RegBus] id_reg2_o;  
wire id_wreg_o;  
wire[`RegAddrBus] id_wd_o;  
  
// 连接ID/EX模块输出与执行阶段EX模块的输入的变量  
wire[`AluOpBus] ex_aluop_i;  
wire[`AluSelBus] ex_alusel_i;  
wire[`RegBus] ex_reg1_i;  
wire[`RegBus] ex_reg2_i;  
wire ex_wreg_i;  
wire[`RegAddrBus] ex_wd_i;  
  
// 连接执行阶段EX模块的输出与EX/MEM模块的输入的变量  
wire ex_wreg_o;  
wire[`RegAddrBus] ex_wd_o;  
wire[`RegBus] ex_wdata_o;  
  
// 连接EX/MEM模块的输出与访存阶段MEM模块的输入的变量  
wire mem_wreg_i;
```

```
wire[`RegAddrBus] mem_wd_i;
wire[`RegBus] mem_wdata_i;

// 连接访存阶段MEM模块的输出与MEM/WB模块的输入的变量
wire mem_wreg_o;
wire[`RegAddrBus] mem_wd_o;
wire[`RegBus] mem_wdata_o;

// 连接MEM/WB模块的输出与回写阶段的输入的变量
wire wb_wreg_i;
wire[`RegAddrBus] wb_wd_i;
wire[`RegBus] wb_wdata_i;

// 连接译码阶段ID模块与通用寄存器Regfile模块的变量
wire reg1_read;
wire reg2_read;
wire[`RegBus] reg1_data;
wire[`RegBus] reg2_data;
wire[`RegAddrBus] reg1_addr;
wire[`RegAddrBus] reg2_addr;

// pc_reg例化
pc_reg pc_reg0(
    .clk(clk), .rst(rst), .pc(pc), .ce(rom_ce_o)
);

assign rom_addr_o = pc; // 指令存储器的输入地址就是pc的值
```

```
// IF/ID模块例化
if_id if_id0(
    .clk(clk), .rst(rst), .if_pc(pc),
    .if_inst(rom_data_i), .id_pc(id_pc_i),
    .id_inst(id_inst_i)
);

// 译码阶段ID模块例化
id id0(
    .rst(rst), .pc_i(id_pc_i), .inst_i(id_inst_i),

    // 来自Regfile模块的输入
    .reg1_data_i(reg1_data), .reg2_data_i(reg2_data),

    // 送到regfile模块的信息
    .reg1_read_o(reg1_read), .reg2_read_o(reg2_read),
    .reg1_addr_o(reg1_addr), .reg2_addr_o(reg2_addr),

    // 送到ID/EX模块的信息
    .aluop_o(id_aluop_o), .alusel_o(id_alusel_o),
    .reg1_o(id_reg1_o), .reg2_o(id_reg2_o),
    .wd_o(id_wd_o), .wreg_o(id_wreg_o)
);

// 通用寄存器Regfile模块例化
regfile regfile1(
    .clk (clk), .rst (rst),
    .we(wb_wreg_i), .waddr(wb_wd_i),
```

```
.wdata(wb_wdata_i), .re1(reg1_read),
.raddr1(reg1_addr), .rdata1(reg1_data),
.re2(reg2_read), .raddr2(reg2_addr),
.rdata2(reg2_data)

);

// ID/EX模块例化
id_ex id_ex0(
    .clk(clk),           .rst(rst),
    // 从译码阶段ID模块传递过来的信息
    .id_aluop(id_aluop_o), .id_alusel(id_alusel_o),
    .id_reg1(id_reg1_o), .id_reg2(id_reg2_o),
    .id_wd(id_wd_o), .id_wreg(id_wreg_o),
    // 传递到执行阶段EX模块的信息
    .ex_aluop(ex_aluop_i), .ex_alusel(ex_alusel_i),
    .ex_reg1(ex_reg1_i), .ex_reg2(ex_reg2_i),
    .ex_wd(ex_wd_i), .ex_wreg(ex_wreg_i)
);

// EX模块例化
ex ex0(
    .rst(rst),
    // 从ID/EX模块传递过来的的信息
    .aluop_i(ex_aluop_i), .alusel_i(ex_alusel_i),
    .reg1_i(ex_reg1_i), .reg2_i(ex_reg2_i),
```

```
.wd_i(ex_wd_i),      .wreg_i(ex_wreg_i),  
  
// 输出到EX/MEM模块的信息  
.wd_o(ex_wd_o),      .wreg_o(ex_wreg_o),  
.wdata_o(ex_wdata_o)  
);  
  
// EX/MEM模块例化  
ex_mem ex_mem0(  
    .clk(clk),      .rst(rst),  
  
    // 来自执行阶段EX模块的信息  
.ex_wd(ex_wd_o),      .ex_wreg(ex_wreg_o),  
.ex_wdata(ex_wdata_o),  
  
    // 送到访存阶段MEM模块的信息  
.mem_wd(mem_wd_i),      .mem_wreg(mem_wreg_i),  
.mem_wdata(mem_wdata_i)  
);  
  
// MEM模块例化  
mem mem0(  
    .rst(rst),  
  
    // 来自EX/MEM模块的信息  
.wd_i(mem_wd_i),      .wreg_i(mem_wreg_i),  
.wdata_i(mem_wdata_i),
```

```
// 送到MEM/WB模块的信息
.wd_o(mem_wd_o), .wreg_o(mem_wreg_o),
.wdata_o(mem_wdata_o)
);

// MEM/WB模块例化
mem_wb mem_wb0(
    .clk(clk),    .rst(rst),

    // 来自访存阶段MEM模块的信息
    .mem_wd(mem_wd_o), .mem_wreg(mem_wreg_o),
    .mem_wdata(mem_wdata_o),

    // 送到回写阶段的信息
    .wb_wd(wb_wd_i), .wb_wreg(wb_wreg_i),
    .wb_wdata(wb_wdata_i)
);

endmodule
```

至此，ori指令的流水线之旅已经结束了，一个原始而简单的五级流水线结构也已经建立了，有读者可能会怀疑区区百十行代码就实现了流水线，是不是太简单了？有这样的怀疑是正常的，的确很简单，但是简单并不代表简陋，不代表错误，流水线实际上并不是大家想象的那么复杂，4.3节，将验证本节实现的流水线能不能正确工作，能不能正确执行ori指令。

4.3 验证OpenMIPS实现效果

4.3.1 指令存储器ROM的实现

本节将验证我们的OpenMIPS是否实现正确，包含：流水线是否正确、ori指令是否实现正确。在验证之前，需要首先实现指令存储器，以便OpenMIPS从中读取指令。

指令存储器ROM模块是只读的，其接口如图4-7所示，还是采用左边是输入接口、右边是输出接口的方式绘制，这样便于理解。接口含义如表4-12所示。

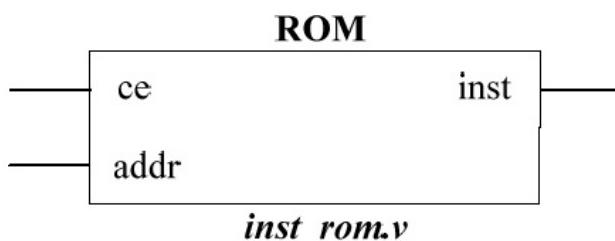


图4-7 指令存储器ROM模块接口图

表4-12 指令存储器ROM模块的接口描述

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	ce	1	输入	使能信号
2	addr	32	输入	要读取的指令地址
3	inst	32	输出	读出的指令

指令存储器ROM模块在文件inst_rom.v中实现，代码如下，可以在本书附带光盘的Code\Chapter4\目录下找到源文件。

```
module inst_rom(  
    input wire      ce,  
    input wire[`InstAddrBus] addr,  
    output reg[`InstBus]  inst
```

```

);

// 定义一个数组，大小是InstMemNum，元素宽度是InstBus
reg[`InstBus] inst_mem[0:`InstMemNum-1];

// 使用文件inst_rom.data初始化指令存储器
initial $readmemh( "inst_rom.data", inst_mem );

// 当复位信号无效时，依据输入的地址，给出指令存储器ROM中对应的元素
always @ (*) begin
    if (ce == `ChipDisable) begin
        inst <= `ZeroWord;
    end else begin
        inst <= inst_mem[addr[`InstMemNumLog2+1:2]];
    end
end

endmodule

```

这个代码很好理解，有以下几点说明。

(1) 在初始化指令存储器时，使用了initial过程语句。initial过程语句只执行一次，通常用于仿真模块中对激励向量的描述，或用于给变量赋初值，是面向模拟仿真过程语句，通常不能被综合工具支持。所以如果要将本章实现的OpenMIPS处理器使用综合工具进行综合，那么需要修改这里初始化指令存储器的方法。

(2) 在初始化指令存储器时，使用了系统函数\$readmemh，表示从inst_rom.data文件中读取数据以初始化inst_mem，而inst_mem正是之前定

义的数组。inst_rom.data是一个文本文件，里面存储的是指令，其每行存储一条32位宽度的指令（使用十六进制表示），系统函数\$readmemh会将inst_rom.data中的数据依次填写到inst_mem数组中。

(3) OpenMIPS是按照字节寻址的，而此处定义的指令存储器的每个地址是一个32bit的字，所以要将OpenMIPS给出的指令地址除以4再使用，比如：要读取地址0xC处的指令，那么实际就是对应ROM的inst_mem[3]，如图4-8所示。

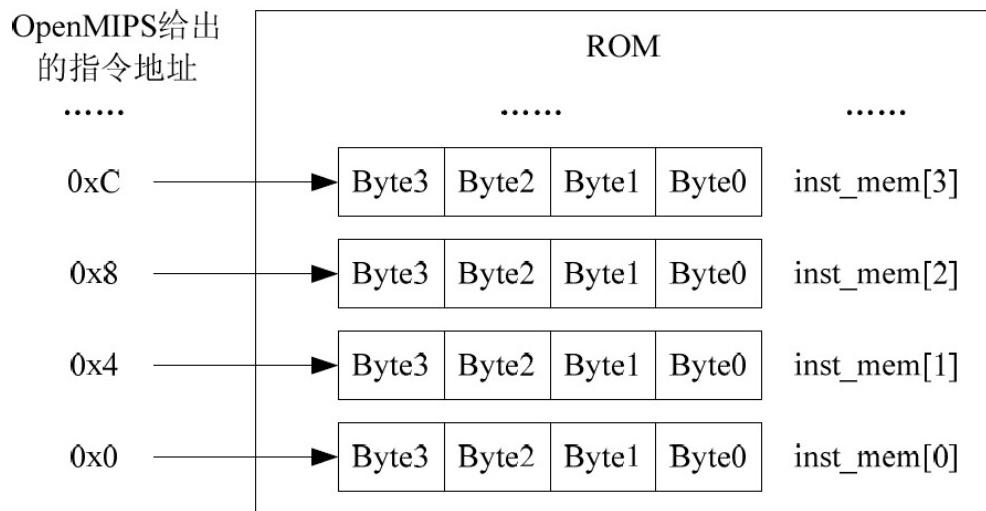


图4-8 OpenMIPS给出的指令地址与ROM中元素位置的关系

除以4也就是将指令地址右移2位，所以在读取的时候给出的地址是addr[`InstMemNumLog2 +1:2]，其中InstMemNumLog2是指令存储器的实际地址宽度，比如：如果inst_mem有1024个元素，那么InstMemNum等于1024，InstMemNumLog2等于10，表示实际地址宽度为10。

4.3.2 最小SOPC的实现

为了验证，需要建立一个SOPC，其中仅OpenMIPS、指令存储器ROM，所以是一个最小SOPC。OpenMIPS从指令存储器中读取指令，指令

进入OpenMIPS开始执行。最小SOPC的结构如图4-9所示。

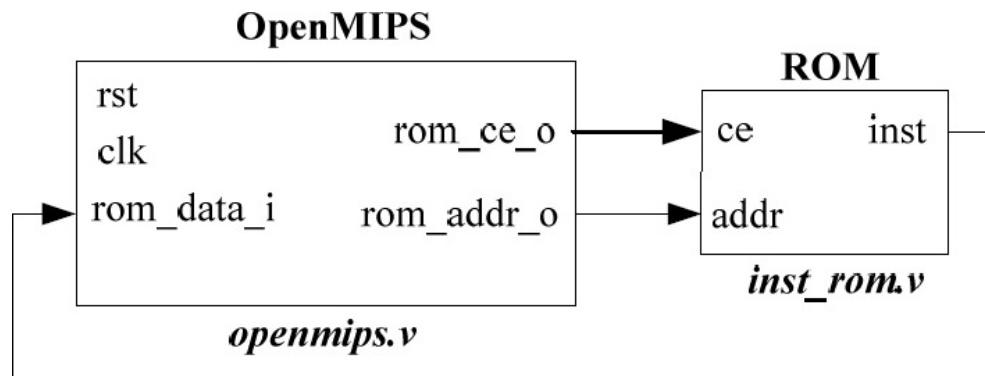


图4-9 最小SOPC的结构

最 小 SOPC 对 应 的 模 块 是 openmips_min_sopc ， 位 于 文 件 openmips_min_sopc.v 中， 读 者 可 以 在 本 书 附 带 光 盘 的 Code\Chapter4\ 目 录 下 找 到 该 文件， 主 要 内 容 如 下。 在 其 中 例 化 了 处 理 器 OpenMIPS 、 指 令 存 储 器 ROM， 并 将 两 者 按 照 图 4-9 的 方 式 进 行 连 接。

```
module openmips_min_sopc(  
  
    input wire clk,  
    input wire rst  
  
);  
  
    // 连接指令存储器  
    wire[`InstAddrBus] inst_addr;  
    wire[`InstBus]      inst;  
    wire                  rom_ce;  
  
    // 例化处理器OpenMIPS  
    openmips openmips0(  
        .clk(clk),  
        .rst(rst),  
        .rom_ce(rom_ce),  
        .rom_addr(inst_addr),  
        .rom_data(inst)  
    );  
endmodule
```

```

        .clk(clk),           .rst(rst),
        .rom_addr_o(inst_addr), .rom_data_i(inst),
        .rom_ce(rom_ce)
    );

    // 例化指令存储器ROM
    inst_rom inst_rom0(
        .ce(rom_ce),
        .addr(inst_addr),   .inst(inst)
    );
}

endmodule

```

4.3.3 编写测试程序

我们需要写一段测试程序，并将其存储到指令存储器ROM，这样当4.3.2节建立的最小SOPC开始运行的时候，就会从ROM中取出我们的程序，送入OpenMIPS处理器执行。由于目前的OpenMIPS只实现了一条ori指令，所以测试程序很简单，如下所示，对应本书附带光盘Code\Chapter4\TestAsm目录下的inst_rom.S文件。

```

ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100
ori $2,$0,0x0020      # $2 = $0 | 0x0020 = 0x0020
ori $3,$0,0xff00      # $3 = $0 | 0xff00 = 0xff00
ori $4,$0,0xffff      # $4 = $0 | 0xffff = 0xffff

```

测试程序共有4条指令，都是ori指令。

第1条指令将0x1100进行零扩展后与寄存器\$0进行逻辑“或”运算，结果保存在寄存器\$1中。

第2条指令将0x0020进行零扩展后与寄存器\$0进行逻辑“或”运算，结果保存在寄存器\$2中。

第3条指令将0xff00进行零扩展后与寄存器\$0进行逻辑“或”运算，结果保存在寄存器\$3中。

第4条指令将0xffff进行零扩展后与寄存器\$0进行逻辑“或”运算，结果保存在寄存器\$4中。

指令的注释说明了指令的执行结果。接下来，按照正常的顺序应该是使用编译器编译我们的测试程序，但由于GCC编译器的安装、使用、Makefile文件的制作等内容还需要不少篇幅讲解，而想必各位读者和笔者一样，急切地想知道OpenMIPS是否实现正确，所以本节采用手工编译的方式编译测试程序，4.4节将专题介绍GCC编译器的使用。

手工编译只须按照指令内容填充进如图4-1所示的ori指令格式中，即可得到对应的二进制字，比如：对于指令ori \$1,\$0,0x1100，对应的二进制字如图4-10所示。

31	26 25	21 20	16 15	0
ORI 001101	00000	00001	0001 0001 0000 0000	

图4-10 指令ori \$1,\$0,0x1100对应的二进制字

转化为十六进制即0x34011100，其余3条指令按照同样的方式可以得到对应的二进制字，按照\$readmemh函数的要求，一行放一条指令，得到测试程序对应的isnt_rom.data文件如下，可在本书附带光盘的Code\Chapter4\TestAsm目录下找到同名文件。

```
34011100  
34020020  
3403ff00  
3404ffff
```

4.3.4 建立Test Bench文件

本节将建立Test Bench文件，其中给出最小SOPC运行所需的时钟信号、复位信号。代码如下，对应本书附带光盘Code\Chapter4\目录下的openmips_min_sopc_tb.v文件。

```
// 时间单位是1ns，精度是1ps  
'timescale 1ns/1ps  
  
module openmips_min_sopc_tb();  
  
reg      CLOCK_50;  
reg      rst;  
  
// 每隔10ns，CLOCK_50信号翻转一次，所以一个周期是20ns，对应50MHz  
initial begin  
    CLOCK_50 = 1'b0;  
    forever #10 CLOCK_50 = ~CLOCK_50;  
end  
  
// 最初时刻，复位信号有效，在第195ns，复位信号无效，最小SOPC开始运行  
// 运行1000ns后，暂停仿真  
initial begin
```

```
rst = `RstEnable;
#195 rst= `RstDisable;
#1000 $stop;
end

// 例化最小SOPC
openmips_min_sopc openmips_min_sopc0(
    .clk(CLOCK_50),
    .rst(rst)
);

endmodule
```

4.3.5 使用ModelSim检验OpenMIPS实现效果

万事俱备，只欠东风了，本节是验证前的最后一步——建立ModelSim工程，进行仿真。参考第2章的介绍，新建一个ModelSim工程，工程名可以为openmips_min_sopc，将上文创建的OpenMIPS所有源文件、Test Bench文件、指令存储器的源文件等（也就是本书附带光盘Code\Chapter4目录下所有.v文件）添加到工程中，然后编译。

注意：还需要将4.3.4节制作的inst_rom.data文件复制到ModelSim工程目录下。

编译通过后，将workspace切换到Library选项卡，打开work这个library，选中openmips_min_sopc_tb，用鼠标右键单击，选择Simulate选项，如图4-11所示。

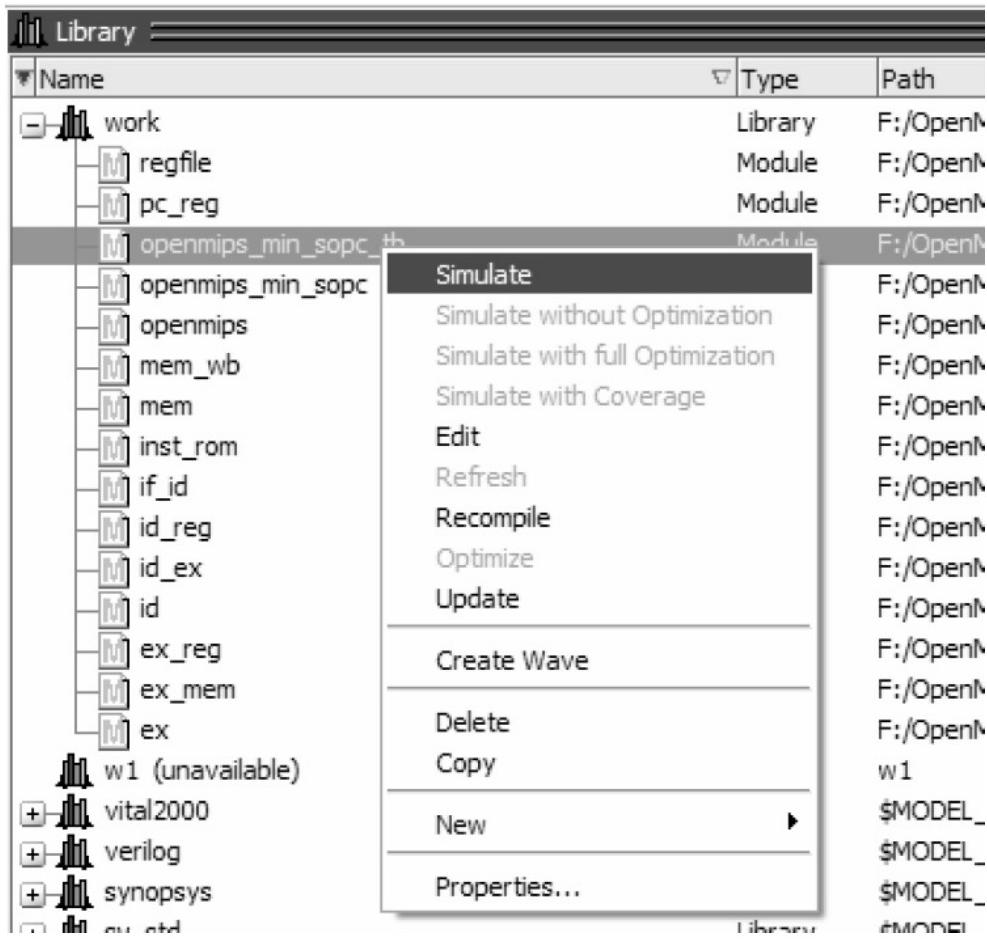


图4-11 选择openmips_min_sopc_tb作为仿真对象

在出现的波形显示界面中，添加要观察的信号，即可开始仿真。此处我们选择寄存器\$1-\$4作为观察对象，如图4-12所示，通过观察寄存器\$1-\$4的最终值，可知OpenMIPS正确执行了测试程序，也就是正确实现了ori指令。

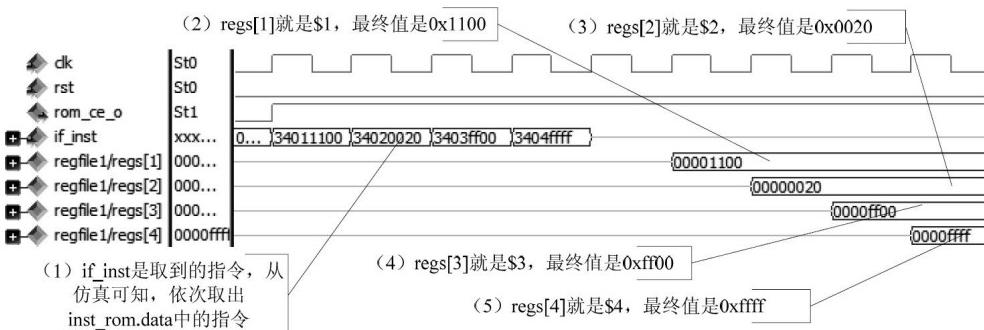


图4-12 从仿真结果可知OpenMIPS正确实现了ori指令

添加更多要观察的信号，可以了解流水线执行情况，如图4-13所示。为了使流水线情况显示得更加直观，此处以第一条指令在流水线中的执行过程为例，并且图中去掉了其他指令执行时引起的信号变化。

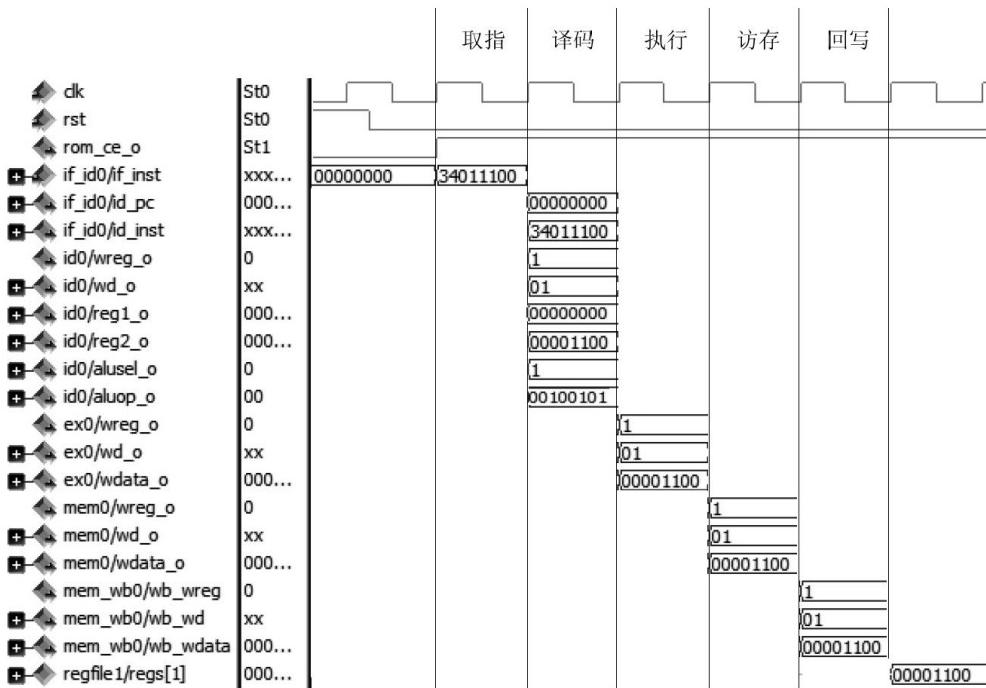


图4-13 通过观察第一条指令的执行过程，判断OpenMIPS五级流水线是否正确实现

(1) 在复位结束后的第一个时钟周期上升沿，rom_ce_o 变为 ChipEnable，表示指令存储器使能，开始取指，进入取指阶段，从指令存储器中取出第一条指令0x34011100，赋给IF/ID模块的输入端口if_inst。下一个时钟周期，第一条指令进入译码阶段。

(2) 观察译码阶段

- 此时译码阶段的指令id_inst正是第一条指令0x34011100。
- 指令地址id_pc是0x00000000。
- 在ID模块对指令进行译码，得到指令运算类型alusel_o是3'b001，查询defines.h文件中的宏定义可知，对应宏EXE_RES_LOGIC，表示是逻辑运算。

- 得到运算子类型aluop_o是8'b00100101，查询defines.h文件中的宏定义可知，对应宏EXE_OR_OP，表示逻辑“或”运算。
- 译码得到参与运算的源操作数1是0x00000000，正是\$0寄存器的值。
- 译码得到参与运算的源操作数2是0x00001100，正是指令中立即数零扩展后的值。
- 译码得到wreg_o的值为1，表示要写目的寄存器。
- 译码得到要写入的目的寄存器wd_o是5'b00001，正是\$1寄存器。

(3) 观察执行阶段

- 进行指定的运算，得到wdata_o为0x00001100，就是要写到目的寄存器的数据。
- 传递译码阶段wreg_o的值，为1，表示要写目的寄存器。
- 传递译码阶段wd_o的值，为5'b00001，表示要写入的目的寄存器是\$1寄存器。

(4) 观察访存阶段

- 传递执行阶段wdata_o的值，为0x00001100，表示要写到目的寄存器的数据。
- 传递执行阶段wreg_o的值，为1，表示要写目的寄存器。
- 传递执行阶段wd_o的值，为5'b00001，表示要写入的目的寄存器是\$1寄存器。

(5) 观察回写阶段

- 得到访存阶段wdata_o的值，为0x00001100，表示要写到目的寄存器的数据。
- 得到访存阶段wreg_o的值，为1，表示要写目的寄存器。

- 得到访存阶段wd_o的值，为5'b00001，表示要写入的目的寄存器是\$1寄存器。

在回写阶段的最后，将按照要求写目的寄存器\$1，使得\$1的值为0x00001100。通过上面的观察，可知原始的OpenMIPS五级流水线实现正确。接下来，我们就可以以此为基础，不断充实，添加实现更多的MIPS指令，不过，在此之前，我们要先学习使用GNU工具链，本节的例子只有4条指令，可以手工编译，以后会遇到比较复杂，拥有较多指令的程序，届时，手工编译就显得效率低下了，所以要使用GNU工具链。

4.4 MIPS编译环境的建立

OpenMIPS处理器在设计的时候就计划与MIPS32指令集架构兼容，所以可以使用MIPS32架构下已有的GNU开发工具链。本节将说明如何安装使用GNU开发工具链以及如何制作Makefile文件，从而以更加方便、快捷、自动的方式对测试程序进行编译，并得到指令存储器ROM的初始化文件inst_rom.data。

4.4.1 VisualBox的安装与设置

GNU工具链要安装在Linux环境下，大多数读者使用的可能都是Windows平台，可以首先安装Linux虚拟机，再在Linux虚拟机中安装GNU工具链。笔者推荐使用OpenCores站点上提供的一个Linux虚拟机镜像，该虚拟机预装的是Ubuntu系统。

在浏览器中输入地址：<ftp://openrisc.opencores.org/virtualbox-image/>，FTP的用户名和密码都是openrisc，登录后会出现如图4-14所示界面。



图4-14 Ubuntu虚拟机镜像下载

下载最新的那个文件就可以了，笔者使用的是2011-12-15版。下载完成后解压该文件，大约4GB左右。此时还需要下载VirtualBox才可以打开该文件。VirtualBox是一款开源的虚拟机软件，本书使用的是4.1.22版。下载完成后安装VirtualBox，安装完成后打开VirtualBox，界面如图4-15所示。



图4-15 VirtualBox主界面

单击“新建”按钮出现“新建虚拟机”向导，单击“下一步”按钮，出现如图4-16所示界面。



图4-16 新建虚拟机设置一

此处操作系统选择Linux，版本选择Ubuntu，设置内存大小，单击下一步按钮，如图4-17所示。



图4-17 新建虚拟机设置二

内存大小依据计算机情况设置，笔者设置的是512MB，已经够用了，毕竟我们需要编译的程序都是十分简单的，单击下一步按钮，选择“使用

现有的虚拟硬盘”，然后选择解压后的虚拟机文件，如图4-18所示。



图4-18 新建虚拟机设置三

单击“下一步”按钮，VisualBox会将用户刚才的设置都列出来，确认无误后，单击“创建”按钮，这样虚拟机就创建好了。启动虚拟机，Ubuntu虚拟机桌面显示如图4-19所示。

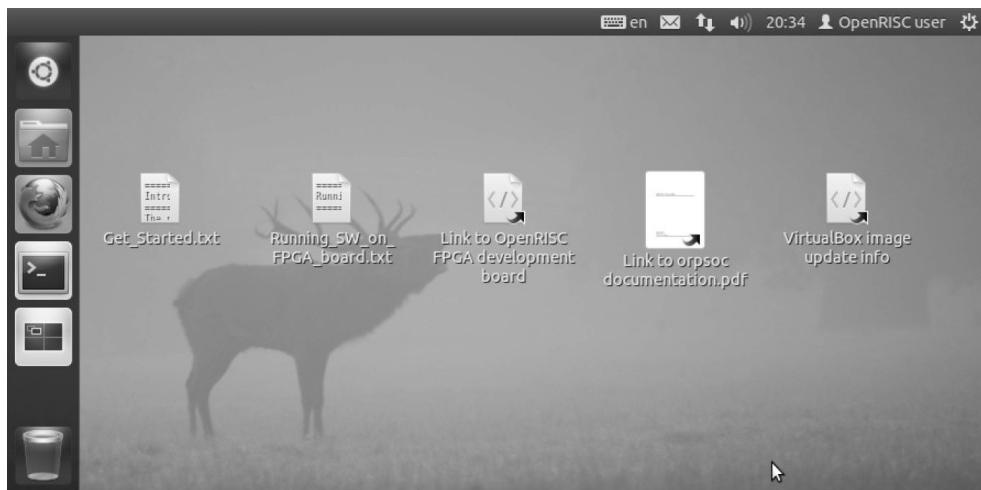


图4-19 Ubuntu虚拟机桌面

至此Linux虚拟机就已经安装好了，还需要多做一步工作，就是设置虚拟机与Windows宿主机之间的共享，这样方便以后在两个系统之间传递文件。先关闭Ubuntu虚拟机，然后打开VisualBox中虚拟机的设置界面，选择“共享文件夹”选项，如图4-20所示。



图4-20 虚拟机与宿主机共享文件夹设置步骤一

单击界面右边的添加文件夹按钮，出现如图4-21所示的界面。



图4-21 虚拟机与宿主机共享文件夹设置步骤二

在其中选择共享文件夹的路径，设置名称，参考图4-21所示设置。设置完成后，可以启动虚拟机，打开终端，输入命令：

```
sudo mount -t vboxsf UbuntuShareFolder /mnt/
```

该命令的作用是将共享文件夹挂载在/mnt/目录下，sudo表示以Root用户身份执行该命令，终端会提示输入密码，Ubuntu虚拟机默认Root用户的密码是openrisc。这样就实现了虚拟机与宿主机的文件共享，对虚拟机而言共享文件放在/mnt/路径下，对宿主机而言共享文件放在图4-21所示的F盘UbuntuShareFolder文件夹下。

4.4.2 GNU工具链的安装

在本书附带光盘的tools目录下提供了GNU工具链安装文件，文件名是mips-sde-elf-i686-pc-linux-gnu.tar.tar，将该文件复制到4.4.1节设置的共享文件夹下，即可通过Ubuntu虚拟机访问该文件。将安装文件复制到Ubuntu的/opt目录下，打开Ubuntu的终端，使用如下命令解压缩：

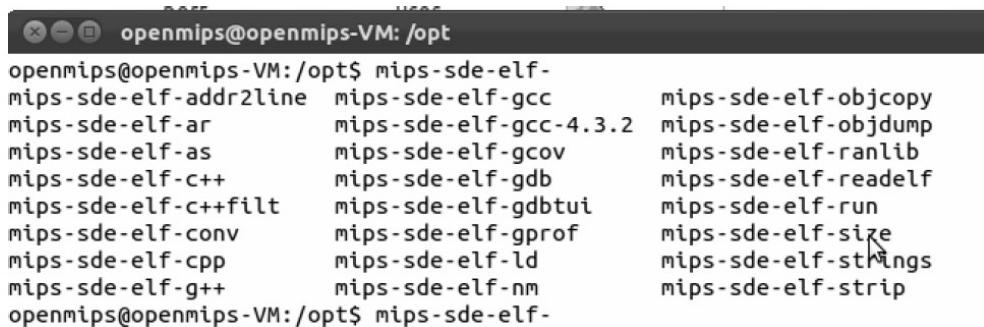
```
cd /opt  
tar vfxj mips-sde-elf-i686-pc-linux-gnu.tar.tar
```

然后打开用户主目录Home文件夹，在窗口菜单栏中选择View->Show Hidden Files，以显示所有文件，这样可以找到一个隐藏文件.bashrc，在此文件的最后加入PATH的设置，如下。

```
export PATH="$PATH:/opt/mips-4.3/bin"
```

重新启动Ubuntu系统。

重启后，打开终端，在其中输入mips-sde-elf-，然后按两次Tab键，会列出刚刚安装的针对MIPS平台的所有编译工具，如图4-22所示，表示GNU工具链安装成功。



```
openmips@openmips-VM: /opt$ mips-sde-elf-
mips-sde-elf-addr2line  mips-sde-elf-gcc      mips-sde-elf-objcopy
mips-sde-elf-ar        mips-sde-elf-gcc-4.3.2  mips-sde-elf-objdump
mips-sde-elf-as        mips-sde-elf-gcov     mips-sde-elf-ranlib
mips-sde-elf-c++       mips-sde-elf-gdb      mips-sde-elf-readelf
mips-sde-elf-c++filt   mips-sde-elf-gdbtui   mips-sde-elf-run
mips-sde-elf-conv      mips-sde-elf-gprof    mips-sde-elf-size
mips-sde-elf-cpp       mips-sde-elf-ld      mips-sde-elf-strengs
mips-sde-elf-g++       mips-sde-elf-nm      mips-sde-elf-strip
openmips@openmips-VM: /opt$ mips-sde-elf-
```

图4-22 Ubuntu中安装的编译工具

GNU工具链包含很多工具，但我们使用得不多，主要的几个工具如下。此处使用的是通用名，针对MIPS平台的工具，会在名称前增加“mips-sde-elf-”前缀。

- as：GNU汇编器，通常也称为GAS (GNU Assembler)，as对汇编源程序进行编译产生目标文件。
- ld：GNU链接器，as产生的目标文件需要由ld进行链接、重定位数据产生可执行文件。
- objcopy：用于将一种格式的目标文件复制成另外一种格式。
- objdump：用于列出关于二进制文件的各种信息。
- readelf：类似于objdump，但是它只能处理ELF格式的文件。

4.4.3 使用GNU工具进行编译

本小节就使用GNU工具编译4.3节的测试程序。首先在Ubuntu虚拟机中新建一个文件，文件名为inst_rom.S，内容如下。相比4.3节的测试程序，多了三条编译指导语句。

```

.org 0x0          // 指示程序从地址0x0开始
.global _start    // 定义一个全局符号_start
.set noat        // 允许自由使用寄存器$1

_start:
    ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100
    ori $2,$0,0x0020      # $2 = $0 | 0x0020 = 0x0020
    ori $3,$0,0xff00      # $3 = $0 | 0xff00 = 0xff00
    ori $4,$0,0xffff      # $4 = $0 | 0xffff = 0xffff

```

对“set noat”做进一步说明，这是一个汇编控制伪操作。在第1章介绍MIPS32架构中的通用寄存器时已经提到MIPS32中的通用寄存器都有约定名称，其用法也遵循一些约定，比如：寄存器\$1，编程时的约定名称为at，一般留给汇编器使用，程序中不直接使用。如果直接使用，汇编器会发出警告，此处设置“set noat”就是表示可以自由使用寄存器\$1，汇编器不会发出警告。

在Ubuntu中打开终端，使用cd命令将路径调整到上述inst_rom.S所在目录，然后使用如下命令编译代码。其中添加了“-mips32”选项，表示按照MIPS32指令集架构进行编译。

```
mips-sde-elf-as -mips32 inst_rom.S -o inst_rom.o
```

上述命令会得到目标代码inst_rom.o。打开inst_rom.o文件，可以发现其最初的四个字节是：0x7F、0x45、0x4C、0x46，这说明inst_rom.o是一个ELF文件，如图4-23所示。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	7F	45	4C	46	01	02	01	00	00	00	00	00	00	00	00	00	; ELF.....
00000010h:	00	01	00	08	00	00	00	01	00	00	00	00	00	00	00	00	;
00000020h:	00	00	00	98	50	00	00	00	34	00	00	00	00	00	28	;	...簇....4....(
00000030h:	00	09	00	06	34	01	11	00	34	02	00	20	34	03	FF	00	;4....4.. 4.. .

图4-23 inst_rom.o的开始部分

为了便于读者理解，下面将简单介绍一下ELF文件，读者朋友如果对这不感兴趣或者希望尽快了解编译链接过程的可以跳过下面的介绍，直接阅读4.4.4节。

ELF（Executable and Linkable Format）可执行链接格式，是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的。ELF目标文件有三种类型。

(1) 可重定位 (Relocatable) 文件：保存着代码和适当的数据，用来和其他Object文件一起创建一个可执行文件或共享文件。

(2) 可执行 (Executable) 文件：保存着一个用来执行的程序，该文件指出了如何来创建程序进程映象。

(3) 共享目标文件：包含了在两种使用环境中链接的代码和数据。首先链接器 (ld) 可以将它和其余可重定位文件和共享目标文件一起处理，生成另外一个目标文件（比如：编译器和链接器把*.o和*.so一起装配成一个*.exe文件）。其次，动态链接器 (Dynamic Linker) 可将它与某个可执行文件以及其他共享目标文件组合在一起创建进程映像（比如：动态加载器把*.exe程序和*.so加载进内存执行）。

无论何种类型的ELF文件，其结构都是相同的。ELF文件由四部分组成：ELF header、Program header table、Sections、Section header table。其最开始的部分就是ELF header，定义如下：

```
#define EI_NIDENT 16
typedef struct{
    unsigned char      e_ident[EI_NIDENT]; //占用16个字节
    Elf32_Half        e_type;           //Elf32_Half表示是2个字
    Elf32_Half        e_machine;       //Elf32_Half表示是2个字
    Elf32_Word        e_version;       //Elf32_Word表示是4个字
    Elf32_SignedWord  e_entry;         //Elf32_SignedWord表示是4个字
    Elf32_SignedWord  e_phoff;         //Elf32_SignedWord表示是4个字
    Elf32_SignedWord  e_shoff;         //Elf32_SignedWord表示是4个字
    Elf32_UInt        e_flags;          //Elf32_UInt表示是4个字
    Elf32_SignedWord  e_shstrndx;     //Elf32_SignedWord表示是4个字
} Elf32_Ehdr;
```

```

    Elf32_Half      e_machine;
    Elf32_word      e_version;           //Elf32_Word表示是4个字节大小
    Elf32_Addr      e_entry;            //Elf32_addr也表示4个字节大小
    Elf32_Off       e_phoff;             //Elf32_Off也表示4个字节大小
    Elf32_Off       e_shoff;
    Elf32_Word      e_flags;
    Elf32_Half      e_ehsize;
    Elf32_Half      e_phentsize;
    Elf32_Half      e_phnum;
    Elf32_Half      e_shentsize;
    Elf32_Half      e_shnum;
    Elf32_Half      e_shstrndx;
}Elf32_Ehdr;

```

开始四个字节是固定不变的：0x7F，紧接着是ELF三个字符的ASCII码，这四个字节表明这个文件是一个ELF文件。此处以inst_rom.o为例，介绍e_ident字段后面字节的含义，参考图4-23。

- e_type是01，表示是可重定位文件。
- e_machine表示运行该程序需要的体系结构，此处为0x08，表示MIPS R3000。
- e_version表示文件版本，此处是1。
- e_entry表示程序的入口地址，此处是0x0。
- e_phoff是Program header table在文件中的偏移量（以字节计数），此处是0x0。

- e_shoff是Section header table在文件中的偏移量（以字节计数），此处为0x98。
- e_flags为保存着相关文件的特定处理器信息，此处为0x50000000，表示MIPS32。
- e_ehsize表示ELF header的大小，此处为0x34。
- e_phentsize表示Program header table中每一个条目（一个Program header）的大小，此处为0x0。
- e_phnum表示Program header table中有多少个条目，此处为0。
- e_shentsize表示Section header table中每一个条目（一个Section header）的大小，此处为0x28。
- e_shnum表示Section header table中有多少个条目，此处为0x09。
- e_shstrndx保存着字符表相关入口的节区头部表索引，此处为0x06。

通过上述解释可以了解到这个文件是一个可重定位（Relocatable）文件，不是可执行文件，同时了解到该文件包含的Program header table、Section header table信息。对inst_rom.o而言，没有Program header table。按照给出的偏移信息，我们可以得到Section header table表的位置，通过Section header table得到每个Section的位置。

当然，按照ELF header的内容以及Section header table，我们可以按图索骥地分析所有Section，但是这样做效率太低，借助于GNU工具链中的mips-sde-elf-readelf，我们可以直接得到Section信息，如图4-24所示。

```
mips-sde-elf-readelf -S inst_rom.o
There are 9 section headers, starting at offset 0x98:
Section Headers:
[Nr] Name           Type      Addr     Off      Size    ES Flg Lk Inf Al
[ 0] .null         NULL      00000000 000000 000000 00 0 0 0 0
[ 1] .text          PROGBITS  00000000 000034 000010 00 AX 0 0 4
[ 2] .data          PROGBITS  00000000 000044 000000 00 WA 0 0 1
[ 3] .bss           NOBITS   00000000 000044 000000 00 WA 0 0 1
[ 4] .reginfo       MIPS_REGINFO 00000000 000044 000018 18 0 0 4
[ 5] .pdr           PROGBITS  00000000 00005c 000000 00 0 0 4
[ 6] .shstrtab      STRTAB   00000000 00005c 00003a 00 0 0 1
[ 7] .symtab        SYMTAB   00000000 000200 000070 10 8 6 4
[ 8] .strtab        STRTAB   00000000 000270 000008 00 0 0 1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```

图4-24 利用工具mips-sde-elf-readelf可以得到所有的Section信息

注意添加“-S”选项。这里列出了9个Section的信息，注意其中的“.text”这个Section，它的起始地址是0x34，长度是0x10，我们列出这个Section的内容如图4-25所示。

00000030h:	00 09 00 06	34 01 11 00	34 02 00 20	34 03 FF 00
00000040h:	34 04 FF FF	00 00 00 1E	00 00 00 00	00 00 00 00

图4-25 Section .text的内容

参考4.3.3节可知，这0x10个字节正是测试程序中的4条指令对应的二进制数字。

4.4.4 使用GNU工具进行链接

通过编译得到了一个可重定位ELF文件，但这个文件还不能执行，需要通过链接转化为可执行文件，然后才能执行。使用链接工具mips-sde-elf-ld完成这项工作，在mips-sde-elf-ld的参数中需要声明一个链接描述脚本，链接描述脚本描述了输入文件的各个Section如何映射到输出文件的各个Section中，并控制输出文件中Section和符号的内存布局。可以通过新建一个Document作为链接描述脚本，文件名为ram.ld，内容如下。

```
MEMORY
{
    ram      : ORIGIN = 0x00000000, LENGTH = 0x00001000
}

SECTIONS
{
    .text :
    {
        *(.text)
    } > ram

    .data :
    {
        *(.data)
    } > ram

    .bss :
    {
        *(.bss)
    } > ram
}

ENTRY (_start)
```

其中定义了一个存储块——ram，其起始地址是0x0，长度是0x1000，然后指示链接器输出文件包含三个Section，分别是.text、.data、.bss，其中.text从ram的起始地址开始存放，后面跟着.data、.bss，并且输入文件的

Section .text存放在输出文件的.text中，输入文件的Section .data存放在输出文件的.data中，输入文件的Section .bss存放在输出文件的.bss中。最后的Entry指定程序的入口地址，也就是第一条执行的指令地址是_start符号的值，从汇编代码中可知_start符号就是0x0。现在就可以使用链接器了，在Ubuntu虚拟机的终端中输入如下命令。

```
mips-sde-elf-ld -T ram.ld inst_rom.o -o inst_rom.om
```

得到链接后的文件inst_rom.om，这也是一个ELF格式的文件，其ELF header如图4-26所示。

```
00000000h: 7F 45 4C 46 01 02 01 00 00 00 00 00 00 00 00 00 ; ELF.....  
00000010h: 00 02 00 08 00 00 00 01 00 00 00 00 00 00 00 34 ; .....4  
00000020h: 00 01 00 54 50 00 00 00 00 34 00 20 00 01 00 28 ; ...TP....4. ....  
00000030h: 00 06 00 03 00 00 00 01 00 01 00 00 00 00 00 00 ; .....
```

图4-26 inst_rom.om的ELF header

上一小节是手工分析inst_rom.o的ELF header，主要是为了帮助读者理解，其实可以直接使用工具分析ELF header，在终端中输入如下命令将自动分析inst_rom.om的ELF header。

```
mips-sde-elf-readelf -h inst_rom.om
```

其中加上参数“-h”表示只读取ELF header，得到结果如图4-27所示。

```

mips-sde-elf-readelf -h inst_rom.om
ELF Header:
  Magic: 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, big endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: MIPS R3000
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 52 (bytes into file)
  Start of section headers: 65620 (bytes into file)
  Flags: 0x50000000, mips32
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 1
  Size of section headers: 40 (bytes)
  Number of section headers: 6
  Section header string table index: 3

```

图4-27 inst_rom.om的ELF header信息

从图4-27中可知inst_rom.om是一个可执行文件。读者朋友也许已经注意到了，inst_rom.om比inst_rom.o多了Program header，而这在inst_rom.o里面是没有的，与Section header一样，Program header也可以使用一个结构体描述，如下。

```

typedef struct{
    Elf32_Word        p_type;
    Elf32_Off         p_offset;
    Elf32_Addr        p_vaddr;
    Elf32_Addr        p_paddr;
    Elf32_Word        p_filez;
    Elf32_Word        p_memsz;
    Elf32_Word        p_flags;
    Elf32_Word        p_align;
}Elf32_Phdr;

```

我们还是使用工具mips-sde-elf-readelf从inst_rom.om中分析出一个Program header，然后结合这个Program header解释上面各个各项的含义。使用如下命令得到Program header的信息。

```
mips-sde-elf-readelf -l inst_rom.om
```

其中加上“-l”参数，表示列出Program header的信息，显示如图4-28所示。

```
mips-sde-elf-readelf -l inst_rom.om
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 1 program headers, starting at offset 52

Program Headers:
Type          Offset      VirtAddr     PhysAddr     FileSiz MemSiz Flg Align
LOAD          0x010000  0x00000000  0x00000000  0x00010  0x00010 R E 0x10000
```

图4-28 Program header信息

借助上图介绍Program header各个字段的含义：

- p_type为LOAD，表示可加载。
- p_offset表示段的第一个字节在文件inst_rom.om中的偏移，此处为0x10000。
- p_vaddr表示段的第一个字节在内存中地址，此处为0。
- p_paddr为0，在物理地址定位有关联的系统中，该成员是为该段的物理地址而保留的。
- p_filesz表示段在文件中的长度，此处为0x10。
- p_memsz表示段在内存中的长度，此处为0x10。
- p_flags为RE，表示可读、可执行。
- p_align为0x10000，根据此项确定段在文件以及内存中如何对齐。

该Program header表示将inst_rom.om文件中从偏移0x10000开始的0x10个字节放置在内存的0x0处，打开inst_rom.om可以发现从偏移0x10000开始的0x10个字节的内容与inst_rom.o中Section .text的内容一样，所以当这个Program Section加载入内存后，会使得内存从地址0x0开始的0x10个字节存放的就是测试程序的4条指令。

分析到这里，读者是不是对编译、链接过程有了比之前更深的了解？其实这些背景知识与OpenMIPS处理器关系不大，但是知道这些有助于理解编译链接的过程。总结一下，编译链接的过程很简单，只需要两步，如下。

```
编译: mips-sde-elf-as -mips32 inst_rom.S -o inst_rom.o  
链接: mips-sde-elf-ld -T ram.ld inst_rom.o -o inst_rom.om
```

4.4.5 得到ROM初始化文件

4.4.4节得到的inst_rom.om是一个ELF格式的可执行文件，与我们希望的指令存储器ROM初始化文件inst_rom.data的格式有很大区别，需要进行格式转化。在GNU工具链中提供了另一个工具mips-sde-elf-objcopy，用于将一种格式的目标文件转化成另外一种格式。在这里，可以使用mips-sde-elf-objcopy得到inst_rom.om的二进制（Binary）形式，使用方法如下。得到的二进制文件inst_rom.bin的内容如图4-29所示。

```
mips-sde-elf-objcopy -O binary inst_rom.om inst_rom.bin
```

从图4-29可以发现，bin文件的内容正是测试程序中4条指令对应的二进制字，现在只需要编写一个小程序将bin文件转化为ModelSim中存储器初始化文件的格式。这个小程序很简单，此处不再列出代码，在本书附带

的光盘中可以找到源程序，程序名为Bin2Mem.exe，使用方法如下。得到的inst_rom.data文件如图4-30所示。

```
./Bin2Mem.exe -f inst_rom.bin -o inst_rom.data
```

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	34	01	11	00	34	02	00	20	34	03	FF	00	34	04	FF	FF

图4-29 inst_rom.bin文件的内容

1	34011100
2	34020020
3	3403ff00
4	3404ffff

图4-30 inst_rom.data文件的内容

好了，现在回忆一下从源代码得到ModelSim仿真时可以使用的指令存储器ROM初始化文件一共需要4步：编译、链接、得到bin文件、格式转化，如下。

```
编译: mips-sde-elf-as -mips32 inst_rom.S -o inst_rom.o  
链接: mips-sde-elf-ld -T ram.ld inst_rom.o -o inst_rom.om  
得到 bin 文件 : mips-sde-elf-objcopy -O binary inst_rom.om  
inst_rom.bin  
格式转化: ./Bin2Mem.exe -f inst_rom.bin -o inst_rom.data
```

4.4.6 编写Makefile文件

为了得到指令存储器初始化文件，我们需要输入4条命令，有点麻烦，最好只输入一条命令就可以了，这需要使用Makefile文件。在汇编程序inst_rom.S所在目录下新建一个Document，文件名为Makefile，内容如下。

```
ifndef CROSS_COMPILE
CROSS_COMPILE = mips-sde-elf-
endif
CC = $(CROSS_COMPILE)as
LD = $(CROSS_COMPILE)ld
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump

OBJECTS = inst_rom.o

export CROSS_COMPILE

# *****
# Rules of Compilation
# *****

all: inst_rom.data

%.o: %.S
    $(CC) -mips32 $< -o $@

inst_rom.rom: ram.ld $(OBJECTS)
    $(LD) -T ram.ld $(OBJECTS) -o $@
```

```
inst_rom.bin: inst_rom.om  
        $(OBJCOPY) -O binary $< $@  
  
inst_rom.data: inst_rom.bin  
        ./Bin2Mem.exe -f $< -o $@  
  
clean:  
        rm -f *.o *.om *.bin *.data
```

这是一个很简单的Makefile，借助于它介绍Makefile的组成。Makefile的前半部分是一些变量的定义，比如：定义CC为mips-sde-elf-as，定义LD为mips-sde-elf-ld，其中引用了预定义的变量CROSS_COMPILE。Makefile的后半部分定义了多个目标，有all、clean等，采用的语法如下：

目标：依赖文件

命令

上述形式表示的意思是：（1）要想得到“目标”，那么需要执行“命令”；（2）“目标”依赖于“依赖文件”，当“依赖文件”中至少一个文件比“目标”文件新时，“命令”才被执行。在上面Makefile的“命令”中使用了Makefile一些预定义的变量，含义如下：

\$< 表示第一个依赖文件的名称

\$@ 表示目标的完整名称

所以上述Makefile可以解读如下：

（1）用户输入make all，要求得到目标all，目标all的依赖文件是inst_rom.data，要先得到inst_rom.data。

(2) 要得到inst_rom.data，需要依赖文件inst_rom.bin。

(3) 要得到inst_rom.bin，需要依赖文件inst_rom.om。

(4) 要得到inst_rom.om，需要依赖文件\$(OBJECTS)，其中OBJECTS是预定义变量，其值为inst_rom.o，所以此处就是需要依赖文件inst_rom.o。

(5) 要得到inst_rom.o，需要依赖文件inst_rom.S，该文件已经提供，满足依赖条件，所以会执行命令\$(CC) -mips \$< -o \$@，实际就是如下命令，执行后得到inst_rom.o。

```
mips-sde-elf-as -mips inst_rom.S -o inst_rom.o
```

(6) 得到inst_rom.o后，满足了目标inst_rom.om的依赖条件，所以可以进一步得到inst_rom.om，通过执行命令\$(LD) -T ram.ld \$(OBJECTS) -o \$@，实际就是通过如下命令得到inst_rom.om。

```
mip-sde-elf-ld -T ram.ld inst_rom.o -o inst_rom.om
```

(7) 得到inst_rom.om后，满足了目标inst_rom.bin的依赖条件，所以可以进一步得到inst_rom.bin，通过执行命令\$(OBJCOPY) -O binary \$< \$@，实际就是通过如下命令得到inst_rom.bin。

```
mip-sde-elf-objcopy -O binary inst_rom.om inst_rom.bin
```

(8) 得到inst_rom.bin后，满足了目标inst_rom.data的依赖条件，所以可以进一步得到inst_rom.data，通过执行命令./Bin2Mem.exe -f \$< -o \$@，实际就是通过如下命令得到inst_rom.data。

```
./Bin2Mem.exe -f inst_rom.bin -o inst_rom.data
```

(9) 得到inst_rom.data，满足了目标all的依赖条件，从而实现目标all。

有了Makefile文件，我们在终端中输入“make all”就可以完成所有的过程了。

简单总结一下从测试程序得到指令存储器ROM初始化文件的步骤如下。

- 编写源代码，当然是汇编代码，文件名为inst_rom.S。
- 复制本节编写的Makefile、Bin2Mem.exe、ram.ld到源代码所在目录。
- 打开终端，路径调整到源代码所在目录，输入“make all”。

经过上述步骤，即可得到能够在ModelSim仿真中使用的指令存储器ROM初始化文件inst_rom.data。

最后，我们可以增加一步，使用工具mips-sde-elf-objdump对inst_rom.om进行反汇编，从而得到指令与其二进制字的对应。如下。

```
mips-sde-elf-objdump -D inst_rom.om > inst_rom.asm
```

得到inst_rom.asm文件，使用记事本打开该文件，内容如图4-31所示。重点是图4-31中使用黑框包围的部分，其对应的就是测试程序的4条指令。每一行分为三列：左边一列是指令地址，中间一列是指令对应的二进制字，右边是汇编指令。注意：这里的指令进行了变化，虽然测试程序都是ori指令，但是这里都改成li指令，li是汇编指令，ori是机器指令，两者是相等的。

```
li rt, immediate => ori rt,$0,immediate
```

```
inst_rom.om:      file format elf32-tradbigmips

Disassembly of section .text:

00000000 <_start>:
 0: 34011100  li  at,0x1100
 4: 34020020  li  v0,0x20
 8: 3403ff00  li  v1,0xff00
 c: 3404ffff  li  a0,0xffff

Disassembly of section .reginfo:

00000000 <_ram_end-0x10>:
 0: 0000001e  Ox1e
 ...
 00000010 <_ram_end>:
  ...

```

图4-31 inst_rom.asm文件的内容

另外，此处的寄存器没有使用\$1、\$2、\$3、\$4这种方式，而是使用了约定命名。MIPS32中通用寄存器的约定命名可以参考第1章的表1-1。有了inst_rom.asm文件，在进行仿真波形分析的时候，有助于将仿真波形中的32bit二进制指令字与汇编程序中的指令对应，便于分析。

可以修改Makefile文件，使得在编译得到inst_rom.data的同时得到反汇编文件inst_rom.asm，具体修改方法不再详述，读者可以参考本书附带光盘的Code\Chapter4\TestAsm目录下的Makefile文件。

4.5 第一条指令实现小结

本章是很重要的一章，而且内容相对比较杂，在此做一小结，主要做了四项工作。

(1) 本章通过实现指令ori，搭建了一个原始的五级流水线结构，这也是OpenMIPS的核心，当然，目前OpenMIPS还只能执行ori指令，后续会

逐步丰富。

(2) 实现了一个用于测试的最小SOPC，仅仅包括处理器OpenMIPS、指令存储器ROM，并编写了Test Bench测试文件。

(3) 在ModelSim中通过仿真验证了ori指令实现的正确性，也验证了OpenMIPS五级流水线实现的正确性。

(4) 详细介绍了从汇编代码编写的测试程序得到仿真中使用的指令存储器初始化文件的过程，同时，利用Makefile简化了这个过程。

第5章 逻辑、移位操作与空指令的实现

第4章建立了原始的OpenMIPS五级流水线结构，但是只实现了一条ori指令，从本章开始，将逐步完善。本章首先讨论了流水线数据相关问题，然后修改OpenMIPS以解决该问题，并在5.3节验证了解决效果。接着对逻辑、移位操作与空指令的指令格式、用法、作用进行了一一说明，在5.5节通过扩展OpenMIPS实现了这些指令，最后编写测试程序，对实现效果进行了检验。

5.1 流水线数据相关问题

我们在第4章实现的五级流水线结构很简单，如果按照“简单即美”(Simple is Beautiful)的标准，那么我们的流水线是美的，但是不完美，因为现实往往是复杂的，一个简单的流水线是解决不了如此多现实问题的，本节探讨的数据相关问题就是其中一个问题。在我们实现逻辑、移位操作等其他指令之前，必须先讨论这个问题，因为这个问题已经影响测试程序的编写了。

流水线中经常有一些被称为“相关”的情况发生，它使得指令序列中下一条指令无法按照设计的时钟周期执行，这些“相关”会降低流水线的性能。流水线中的相关分为以下三种类型。

(1) 结构相关：指的是在指令执行的过程中，由于硬件资源满足不了指令执行的要求，发生硬件资源冲突而产生的相关。比如：指令和数据都共享一个存储器，在某个时钟周期，流水线既要完成某条指

令对存储器中数据的访问操作，又要完成后续的取指令操作，这样就会发生存储器访问冲突，产生结构相关。

(2) 数据相关：指的是在流水线中执行的几条指令中，一条指令依赖于前面指令的执行结果。

(3) 控制相关：指的是流水线中的分支指令或者其他需要改写PC的指令造成的关系。

结构相关、控制相关将在后续指令分析中讨论，本节重点讨论数据相关的问题。流水线数据相关又分为三种情况：RAW、WAR、WAW。

- RAW，即Read After Write，假设指令j是在指令i后面执行的指令，RAW表示指令i将数据写入寄存器后，指令j才能从这个寄存器读取数据。如果指令j在指令i写入寄存器前尝试读出该寄存器的内容，将得到不正确的数据。
- WAR，即Write After Read，假设指令j是在指令i后面执行的指令，WAR表示指令i读出数据后，指令j才能写这个寄存器。如果指令j在指令i读出数据前就写该寄存器，将使得指令i读出的数据不正确。
- WAW，即Write After Write，假设指令j是在指令i后面执行的指令，WAW表示指令i将数据写入寄存器后，指令j才能将数据写入这个寄存器。如果指令j在指令i之前写该寄存器，将使得该寄存器的值不是最新值。

对于第4章建立的原始OpenMIPS五级流水线而言，从ori指令的实现过程可以知道，只有在流水线回写阶段才会写寄存器（实际上，其

余指令也是一样的，在后面实现其余指令时，对这一点会更加清楚），因此不存在WAW相关。又因为只能在流水线译码阶段读寄存器、回写阶段写寄存器，不存在WAR相关，所以OpenMIPS的流水线只存在RAW相关。RAW相关有三种情况。

① 相邻指令间存在数据相关

考虑如下代码。

```
1 ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100  
2 ori $2,$1,0x0020      # $2 = $1 | 0x0020 = 0x1120
```

第1条ori指令将写寄存器\$1，随后的第2条ori指令需要读出\$1的数据，但是第1条ori指令在回写阶段才会将其运算结果写入\$1，而第2条ori指令在译码阶段就需要读取\$1的值，此时第1条ori指令还处于执行阶段，所以得到的必然不是第1条ori指令计算得出的结果，按这个值运算，必然会出现错误。如图5-1所示，这种情况可以称为相邻指令间存在数据相关，针对OpenMIPS的具体情况，也可以称为流水线译码、执行阶段存在数据相关。

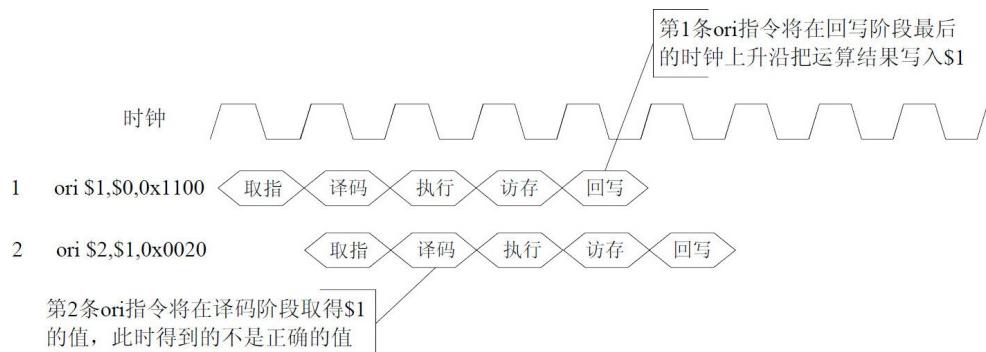


图5-1 相邻指令间存在数据相关

② 相隔1条指令的指令间存在数据相关

考虑如下代码。

```
1 ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100  
2 ori $3,$0,0xffff      # $3 = $0 | 0xffff = 0xffff  
3 ori $2,$1,0x0020      # $2 = $1 | 0x0020 = 0x1120
```

第1条ori指令将写寄存器\$1，第3条ori指令在译码阶段需要读取寄存器\$1，此时第1条ori指令还处于访存阶段，所以得到的必然也不是正确的值。如图5-2所示，这种情况可以称为相隔1条指令的指令间存在数据相关，针对OpenMIPS的具体情况，也可以称为流水线译码、访存阶段存在数据相关。

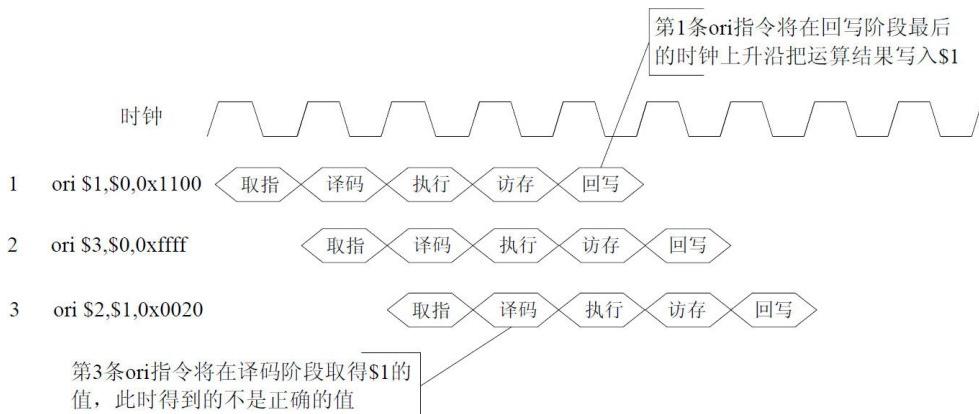


图5-2 相隔1条指令的指令间存在数据相关

③ 相隔2条指令的指令间存在数据相关

考虑如下代码。

```
1 ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100  
2 ori $3,$0,0xffff      # $3 = $0 | 0xffff = 0xffff  
3 ori $4,$0,0xffff      # $4 = $0 | 0xffff = 0xffff  
4 ori $2,$1,0x0020      # $2 = $1 | 0x0020 = 0x1120
```

第1条ori指令将写寄存器\$1，第4条ori指令在译码阶段需要读取寄存器\$1，此时第1条指令处于回写阶段，在回写阶段最后的时钟上升沿才会将运算结果写入\$1，所以第4条ori指令得到的不是正确的寄存器\$1的值。如图5-3所示，这种情况可以称为相隔2条指令的指令间存在数据相关，针对OpenMIPS的具体情况，也可以称为流水线译码、回写阶段存在数据相关。

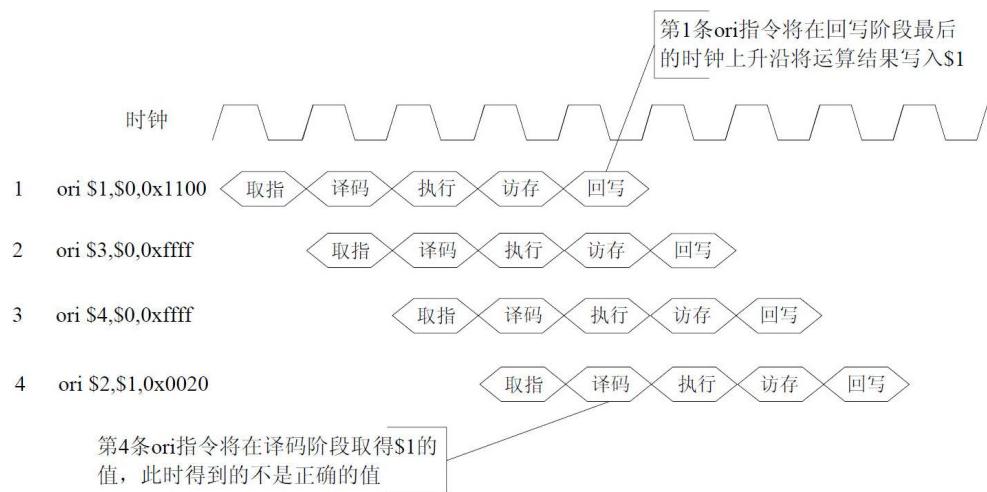


图5-3 相隔2条指令的指令间存在数据相关

其中，相隔2条指令存在数据相关（即流水线译码、回写阶段存在数据相关）这种情况，在第4章设计的Regfile模块中已经得到了解决，Regfile模块部分代码如下。

```
module regfile(  
    .....  
) ;  
  
    .....  
  
/******
```

```
***  
***** 第三段：读端口1的读操作 *****  
*****  
**/  
  
// raddr1是读地址、waddr是写地址、we是写使能、wdata是要写入的数据  
always @ (*) begin  
  
    .....  
  
end else  
  
if((raddr1 == waddr) && (we == `WriteEnable)  
  
&& (re1 == `ReadEnable)) begin  
  
    rdata1 <= wdata;
```

```
    ....  
end  
  
/**********************************************************/  
**  
***** 第四段：读端口2的读操作 *****  
*****  
**/  
  
// raddr2是读地址、waddr是写地址、we是写使能、wdata是要写入的数据  
always @ (*) begin  
  
    ....  
  
    end else  
if((raddr2 == waddr) && (we == `WriteEnable)  
    && (re2 == `ReadEnable)) begin
```

```
rdata2 <= wdata;
```

```
.....
```

```
end
```

```
endmodule
```

在读操作中有一个判断，如果要读取的寄存器是在下一个时钟上升沿要写入的寄存器，那么就将要写入的数据直接作为结果输出。如此就解决了相隔2条指令存在数据相关的情况。

对于相邻指令间存在数据相关、相隔1条指令的指令间存在数据相关这两种情况，有三种解决方法。

① 插入暂停周期：当检测到相关时，在流水线中插入一些暂停周期，如图5-4所示。

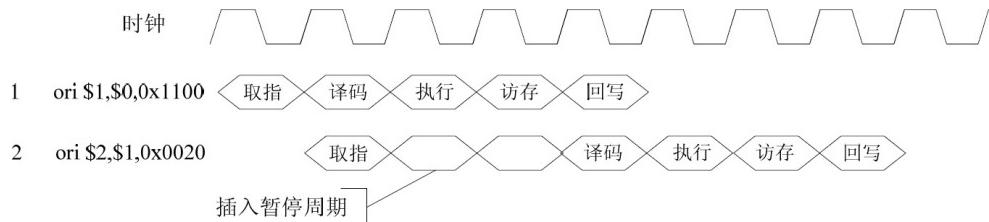


图5-4 在流水线中插入暂停周期消除数据相关

② 编译器调度：编译器检测到相关后，可以改变部分指令的执行顺序，如图5-5所示。

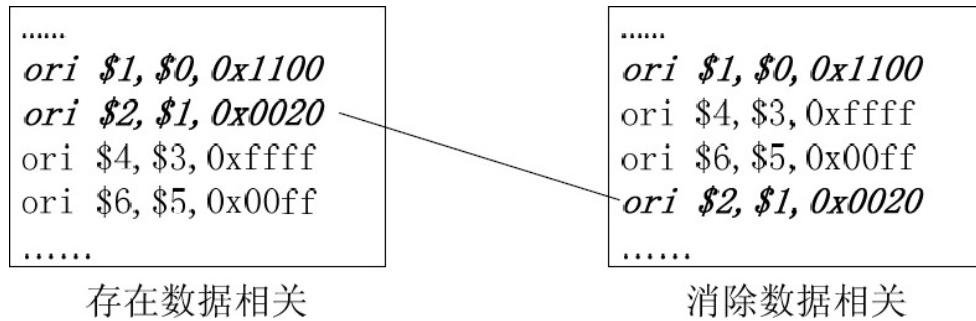


图5-5 编译器通过改变指令执行顺序消除相关

③ 数据前推：将计算结果从其产生处直接送到其他指令需要处或所有需要的功能单元处，避免流水线暂停。如图5-6所示的例子中，新的\$1值实际上在第1条ori指令的执行阶段已经计算出来了，可以直接将该值从第1条ori指令的执行阶段送入第2条ori指令的译码阶段，从而使第2条ori指令在译码阶段得到\$1的新值。也可以直接将该值从第1条ori指令的访存阶段送入第3条ori指令的译码阶段，从而使第3条ori指令在译码阶段也得到\$1的新值。

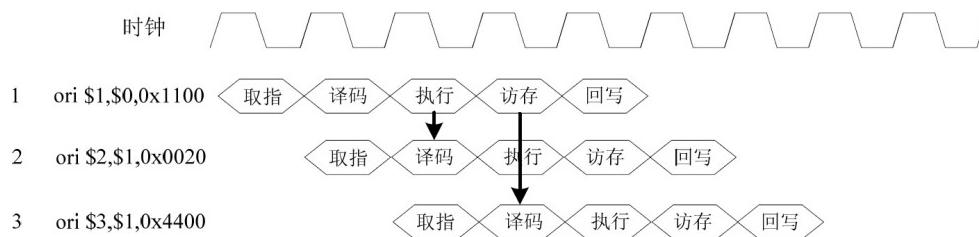


图5-6 数据前推解决流水线相关

读者需要注意，第③种方法有一个前提就是新的寄存器的值可以在执行阶段计算出来，如果是加载指令，那么就不满足这个前提，因

为加载指令在访存阶段才能获得最终结果，这是一种load相关，本书将在实现加载存储指令的时候考虑这种情况，本章暂不考虑。

5.2 OpenMIPS对数据相关问题的解决措施

OpenMIPS处理器采用数据前推的方法来解决流水线数据相关问题。通过补充完善图4-4原始的数据流图，添加部分信号使得可以完成数据前推的工作，如图5-7所示。主要是将执行阶段的结果、访存阶段的结果前推到译码阶段，参与译码阶段选择运算源操作数的过程。

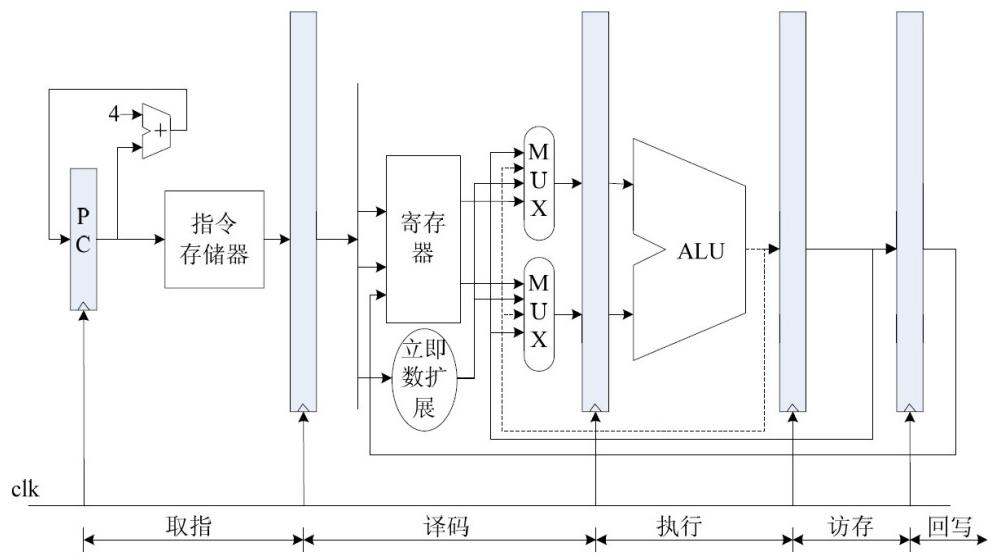


图5-7 添加了数据前推的OpenMIPS数据流图

图5-8给出了为实现数据前推而对OpenMIPS系统结构所做的修改，具体有两个方面。

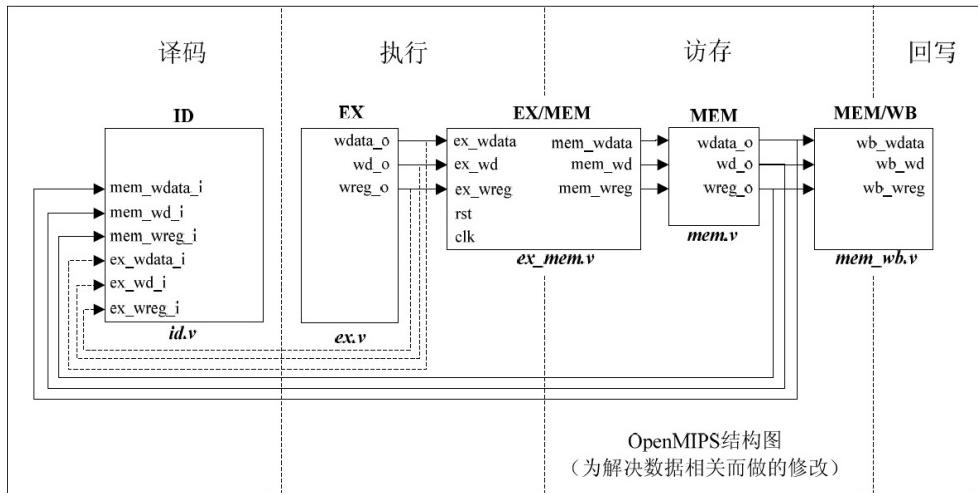


图5-8 为实现数据前推而对OpenMIPS结构所做的修改

(1) 将处于流水线执行阶段的指令的运算结果，包括：是否要写目的寄存器wreg_o、要写的目的寄存器地址wd_o、要写入目的寄存器的数据wdata_o等信息送到译码阶段，如图5-8中虚线所示。

(2) 将处于流水线访存阶段的指令的运算结果，包括：是否要写目的寄存器wreg_o、要写的目的寄存器地址wd_o、要写入目的寄存器的数据wdata_o等信息送到译码阶段。

为此，译码阶段的ID模块要增加如表5-1所示的接口。

表5-1 ID模块要增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	<i>ex_wreg_i</i>	1	输入	处于执行阶段的指令是否要写目的寄存器
2	<i>ex_wd_i</i>	5	输入	处于执行阶段的指令要写的目的寄存器地址
3	<i>ex_wdata_i</i>	32	输入	处于执行阶段的指令要写入目的寄存器的数据
4	<i>mem_wreg_i</i>	1	输入	处于访存阶段的指令是否要写目的寄存器
5	<i>mem_wd_i</i>	5	输入	处于访存阶段的指令要写的目的寄存器地址
6	<i>mem_wdata_i</i>	32	输入	处于访存阶段的指令要写入目的寄存器的数据

译码阶段的ID模块会依据送入的信息，进行综合判断，解决数据相关，给出最后要参与运算的操作数。ID模块的代码要做如下修改，

其中主要的修改部分使用加粗、斜体表示。修改后的代码位于本书光盘中Code\Chapter5_1目录下的id.v文件。

```
module id(
    . . . .
    //处于执行阶段的指令的运算结果

    input wire          ex_wreg_i,
    input wire[`RegBus]   ex_wdata_i,
    input wire[`ReqAddrBus] ex_wd_i,
```

```
//处于访存阶段的指令的运算结果

 mem_wreg_i,

 mem_wdata_i,

 mem_wd_i,

.....
//送到执行阶段的源操作数1、源操作数2
 reg1_o,
 reg2_o,
.....
);
```

```
.....  
  
//给reg1_o赋值的过程增加了两种情况：  
//1. 如果Regfile模块读端口1要读取的寄存器就是执行阶段要写的目的寄  
存器，  
//    那么直接把执行阶段的结果ex_wdata_i作为reg1_o的值；  
//2. 如果Regfile模块读端口1要读取的寄存器就是访存阶段要写的目的寄  
存器，  
//    那么直接把访存阶段的结果mem_wdata_i作为reg1_o的值；  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        reg1_o <= `ZeroWord;  
    end else  
if((reg1_read_o == 1'b1) && (ex_wreg_i == 1'b1)  
&& (ex_wd_i == reg1_addr_o)) begin  
  
    reg1_o <= ex_wdata_i;
```

```
        end else
if((reg1_read_o == 1'b1) && (mem_wreg_i == 1'b1)
&& (mem_wd_i == reg1_addr_o)) begin
    reg1_o <= mem_wdata_i;

    end else if(reg1_read_o == 1'b1) begin
        reg1_o <= reg1_data_i;
    end else if(reg1_read_o == 1'b0) begin
        reg1_o <= imm;
    end else begin
```

```
    reg1_o <= `ZeroWord;
end
end

//给reg2_o赋值的过程增加了两种情况：
//1. 如果Regfile模块读端口2要读取的寄存器就是执行阶段要写的目的寄
存器，
//    那么直接把执行阶段的结果ex_wdata_i作为reg2_o的值；
//2. 如果Regfile模块读端口2要读取的寄存器就是访存阶段要写的目的寄
存器，
//    那么直接把访存阶段的结果mem_wdata_i作为reg2_o的值；

always @ (*) begin
    if(rst == `RstEnable) begin
        reg2_o <= `ZeroWord;
    end else
if((reg2_read_o == 1'b1) && (ex_wreg_i == 1'b1)
&& (ex_wd_i == reg2_addr_o)) begin
```

```
reg2_o <= ex_wdata_i;

    end else

if((reg2_read_o == 1'b1) && (mem_wreg_i == 1'b1)

&& (mem_wd_i == reg2_addr_o)) begin

    reg2_o <= mem_wdata_i;

end else if(reg2_read_o == 1'b1) begin

    reg2_o <= reg2_data_i;

end else if(reg2_read_o == 1'b0) begin

    reg2_o <= imm;

end else begin

    reg2_o <= `ZeroWord;

end

end
```

```
endmodule
```

除了修改译码阶段ID模块的代码，还要修改顶层模块OpenMIPS对应的代码，在其中增加图5-8所示的连接关系。具体修改过程不在书中列出，读者可以参考本书附带光盘中Code\Chapter5_1目录下的openmips.v文件。

5.3 测试数据相关问题的解决效果

测试程序如下，其中存在5.1节讨论的RAW相关的三种情况，源文件是本书附带光盘中Code\Chapter5_1\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.global _start
.set noat
_start:
    ori $1,$0,0x1100      # $1 = $0 | 0x1100 = 0x1100
    ori $1,$1,0x0020      # $1 = $1 | 0x0020 = 0x1120
    ori $1,$1,0x4400      # $1 = $1 | 0x4400 = 0x5520
    ori $1,$1,0x0044      # $1 = $1 | 0x0044 = 0x5564
```

指令的注释给出了预期执行效果。将上述inst_rom.S文件与第4章实现的Bin2Mem.exe、Makefile、ram.ld这三个文件复制到Ubuntu虚拟机中的同一个目录下，打开终端，使用cd命令进入该目录，然后输入make all，即可得到能够用于ModelSim仿真的inst_rom.data文件。

在 ModelSim 中新建一个工程，添加本书附带光盘中 Code\Chapter5_1 目录下的所有.v文件，然后可以编译。再将上面得到的 inst_rom.data 文件复制到 ModelSim 工程的目录下，就可以进行仿真了。ModelSim 中新建工程、仿真的详细步骤可以参考第 2 章。

运行仿真，观察寄存器 \$1 值的变化，如图 5-9 所示，\$1 的变化符合预期，所以修改后的 OpenMIPS 正确解决了数据相关问题。



图 5-9 ModelSim 仿真结果，显示 \$1 的变化符合预期

5.4 逻辑、移位操作与空指令说明

MIPS32 指令集架构中定义的逻辑操作指令有 8 条：and、andi、or、ori、xor、xori、nor、lui，其中 ori 指令已经实现了本章要实现的其余 7 条指令。

MIPS32 指令集架构中定义的移位操作指令有 6 条：sll、sllv、sra、srav、srl、srlv。

MIPS32 指令集架构中定义的空指令有 2 条：nop、ssnop。其中 ssnop 是一种特殊类型的空操作，在每个周期发射多条指令的 CPU 中，使用 ssnop 指令可以确保单独占用一个发射周期。OpenMIPS 设计为标量处理器，也就是每个周期发射一条指令，所以 ssnop 的作用与 nop 相同，可以按照 nop 指令的处理方式来处理 ssnop 指令。

另外，MIPS32指令集架构中还定义了sync、pref这2条指令，其中sync指令用于保证加载、存储操作的顺序，对于OpenMIPS而言，是严格按照指令顺序执行的，加载、存储操作也是按照顺序进行的，所以可以将sync指令当作nop指令处理，在这里将其归纳为空指令。pref指令用于缓存预取，OpenMIPS没有实现缓存，所以也可以将pref指令当作nop指令处理，此处也将其归纳为空指令。

以上17条指令按照格式、作用的不同，又可分为几类，分别说明如下。

1. and、 or、 xor、 nor

这4条指令的格式如图5-10所示，从图中可以发现，这4条指令都是R类型指令，并且指令码都是6'b000000，也就是MIPS32指令集架构中定义的SPECIAL类。此外，第6~10bit都为0，需要依据指令中第0~5bit功能码的值进一步判断是哪一种指令。

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000		rs	rt	rd	00000	AND 100100		and指令
SPECIAL 000000		rs	rt	rd	00000	OR 100101		or指令
SPECIAL 000000		rs	rt	rd	00000	XOR 100110		xor指令
SPECIAL 000000		rs	rt	rd	00000	NOR 100111		nor指令

图5-10 and、 or、 xor、 nor指令格式

- 当功能码是6'b100100时，表示是and指令，逻辑“与”运算。

指令用法为： and rd, rs, rt。

指令作用为： $rd \leftarrow rs \text{ AND } rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行逻辑“与”运算，运算结果保存到地址为rd的通用寄存器中。

- 当功能码是6'b100101时，表示是or指令，逻辑“或”运算。

指令用法为：or rd, rs, rt。

指令作用为： $rd \leftarrow rs \text{ OR } rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行逻辑“或”运算，运算结果保存到地址为rd的通用寄存器中。

- 当功能码是6'b100110时，表示是xor指令，异或运算。

指令用法为：xor rd, rs, rt。

指令作用为： $rd \leftarrow rs \text{ XOR } rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行逻辑“异或”运算，运算结果保存到地址为rd的通用寄存器中。

- 当功能码是6'b100111时，表示是nor指令，或非运算。

指令用法为：nor rd, rs, rt。

指令作用为： $rd \leftarrow rs \text{ NOR } rt$ ，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值进行逻辑“或非”运算，运算结果保存到地址为rd的通用寄存器中。

2. andi、xori指令

这2条指令的格式如图5-11所示，从图5-11中可以发现这2条指令都是I类型指令，可以依据指令中第26~31bit指令码的值判断是哪一种指令。

31	26 25	21 20	16 15	0	
ANDI 001100	rs	rt	immediate		andi指令
XORI 001110	rs	rt	immediate		xori指令

图5-11 andi、xori指令格式

- 当指令码是6'b001100，表示是andi指令，逻辑“与”运算。

指令用法为：andi rt, rs, immediate。

指令作用为： $rt \leftarrow rs \text{ AND zero_extended(immediate)}$ ，将地址为rs的通用寄存器的值与指令中立即数进行零扩展后的值进行逻辑“与”运算，运算结果保存到地址为rt的通用寄存器中。

- 当指令码是6'b001110，表示是xori指令，异或运算。

指令用法为：xori rt, rs, immediate。

指令作用为： $rt \leftarrow rs \text{ XOR zero_extended(immediate)}$ ，将地址为rs的通用寄存器的值与指令中立即数进行零扩展后的值进行逻辑“异或”运算，运算结果保存到地址为rt的通用寄存器中。

3. lui指令

lui指令的格式如图5-12所示，从图中可以发现lui指令是I类型指令，可以依据指令中第26~31bit指令码的值是否为6'b001111，从而判断是否是lui指令。

31	26 25	21 20	16 15	0
LUI 001111	00000	rt	immediate	

图5-12 lui指令格式

指令用法为：lui rt, immediate。

指令作用为： $rt \leftarrow \text{immediate} \parallel 0^{16}$ ，将指令中的16bit立即数保存到地址为rt的通用寄存器的高16位。另外，地址为rt的通用寄存器的低16位使用0填充。

4. sll、sllv、sra、srav、srl、srlv指令

这6条指令的格式如图5-13所示，从图5-13中可以发现这6条指令都是R类型指令，并且指令码都是6'b000000。也就是说，都是SPECIAL类，需要依据指令中第0~5bit功能码的值进一步判断是哪一种指令。

							0
31	26 25	21 20	16 15	11 10	6 5		
SPECIAL 000000	00000	rt	rd	sa	SLL 000000		sll指令
SPECIAL 000000	00000	rt	rd	sa	SRL 000010		srl指令
SPECIAL 000000	00000	rt	rd	sa	SRA 000011		sra指令
SPECIAL 000000	rs	rt	rd	00000	SLLV 000100		sllv指令
SPECIAL 000000	rs	rt	rd	00000	SRLV 000110		srlv指令
SPECIAL 000000	rs	rt	rd	00000	SRAV 000111		sraV指令

图5-13 sll、srl、sra、sllv、srlv、sraV指令格式

- 当功能码是6'b000000，表示是sll指令，逻辑左移。

指令用法为：sll rd, rt, sa。

指令作用为：rd <- rt << sa (logic)，将地址为rt的通用寄存器的值向左移sa位，空出来的位置使用0填充，结果保存到地址为rd的通用寄存器中。

- 当功能码是6'b000010，表示是srl指令，逻辑右移。

指令用法为：srl rd, rt, sa。

指令作用为：rd <- rt >> sa (logic)，将地址为rt的通用寄存器的值向右移sa位，空出来的位置使用0填充，结果保存到地址为rd的通用寄存器中。

- 当功能码是6'b000011，表示是sra指令，算术右移。

指令用法为：sra rd, rt, sa。

指令作用为： $rd \leftarrow rt >> sa$ (arithmetic)，将地址为rt的通用寄存器的值向右移sa位，空出来的位置使用rt[31]的值填充，结果保存到地址为rd的通用寄存器中。

- 当功能码是6'b000100，表示是sllv指令，逻辑左移。

指令用法为：sllv rd, rt, rs。

指令作用为： $rd \leftarrow rt << rs[4:0]$ (logic)，将地址为rt的通用寄存器的值向左移位，空出来的位置使用0填充，结果保存到地址为rd的通用寄存器中。移位位数由地址为rs的寄存器值的第0~4bit确定。

- 当功能码是6'b000110，表示是srlv指令，逻辑右移。

指令用法为：srlv rd, rt, rs。

指令作用为： $rd \leftarrow rt >> rs[4:0]$ (logic)，将地址为rt的通用寄存器的值向右移位，空出来的位置使用0填充，结果保存到地址为rd的通用寄存器中。移位位数由地址为rs的寄存器值的第0~4bit确定。

- 当功能码是6'b000111，表示是srav指令，算术右移。

指令用法为：srav rd, rt, rs。

指令作用为： $rd \leftarrow rt >> rs[4:0]$ (arithmetic)，将地址为rt的通用寄存器的值向右移位，空出来的位置使用rt[31]填充，结果保存到地址为rd的通用寄存器中。移位位数由地址为rs的寄存器值的第0~4bit确定。

总结来说，这6条移位操作指令可以分为两种情况：sllv、srav、srlv这3条指令的助记符最后有“v”，表示移位位数是通过寄存器的值确

定的， sll、 sra、 srl这3条指令的助记符最后没有“v”， 表示移位位数就是指令中第6~10bit的sa值。

5. nop、 ssnop、 sync、 pref指令

这4条指令的格式如图5-14所示。从图5-14中可以发现nop、 ssnop、 sync这3条指令都是R类型指令，并且指令码都是6'b000000，也就是说，都是SPECIAL类。

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000	00000	00000	00000	00000	SLL 000000			nop指令
SPECIAL 000000	00000	00000	00000	00001	SLL 000000			ssnop指令
SPECIAL 000000	00000	00000	00000	00001	SYNC 001111			sync指令
PREF 110011	base	hint		offset				pref指令

图5-14 nop、 ssnop、 sync、 pref指令的格式

更进一步可以发现，nop、 ssnop两条指令的功能码都是6'b000000，与之前介绍的逻辑左移指令sll的功能码相同，这样在译码的时候会不会有冲突：nop指令的二进制码与sll \$0,\$0,0的二进制码一样，处理器如何译码？ssnop指令的二进制码与sll \$0,\$0,1的二进制码一样，处理器如何译码？

```
nop      =      sll $0,$0,0
ssnop    =      sll $0,$0,1
```

其实两者是等价的，sll指令向\$0寄存器保存移位结果，实际不会有任何效果，因为无论向\$0写任何数，其值始终为0，所以效果等同于

什么都不做，这也正是空指令nop、ssnop的效果。所以nop、ssnop指令不用特意实现，完全可以当作特殊的逻辑左移指令sll。

5.5 修改OpenMIPS以实现逻辑、移位操作与空指令

为了实现逻辑、移位操作与空指令（其中nop、ssnop不用特意实现，可以认为是特殊的逻辑左移指令sll），只需要修改OpenMIPS的如下两个模块。

- 修改译码阶段的ID模块，用以实现对上述指令的译码。
- 修改执行阶段的EX模块，使其按照译码结果进行运算。

5.5.1 修改译码阶段的ID模块

首先给出如下宏定义，都在文件defines.v中定义，读者可以在本书附带光盘中Code\Chapter5_2目录下找到该文件。

```
`define EXE_AND 6'b100100          //and指令的功能码
`define EXE_OR   6'b100101          //or指令的功能码
`define EXE_XOR  6'b100110          //xor指令的功能码
`define EXE_NOR  6'b100111          //nor指令的功能码
`define EXE_ANDI 6'b001100          //andi指令的指令码
`define EXE_ORI  6'b001101          //ori指令的指令码
`define EXE_XORI 6'b001110          //xori指令的指令码
`define EXE_LUI  6'b001111          //lui指令的指令码
```

```

`define EXE_SLL  6'b000000          //sll指令的功能码
`define EXE_SLLV 6'b000100          //sllv指令的功能码
`define EXE_SRL  6'b000010          //sra指令的功能码
`define EXE_SRLV 6'b000110          //srlv指令的功能码
`define EXE_SRA  6'b000011          //sra指令的功能码
`define EXE_SRAV 6'b000111          //srav指令的功能码

`define EXE_SYNC 6'b001111          //sync指令的功能码
`define EXE_PREF 6'b110011          //pref指令的指令码
`define EXE_SPECIAL_INST 6'b000000 //SPECIAL类指令的指令码

```

对指令进行译码的前提是能判断出指令种类，这个过程如图5-15所示。其中op就是指令的第26~31bit，即指令码，op2就是指令的第6~10 bit，op3就是指令的第0~5bit，即功能码，op4就是指令的第16~20bit，定义如下。

```

wire[5:0] op  = inst_i[31:26];      // 指令码
wire[4:0] op2 = inst_i[10:6];
wire[5:0] op3 = inst_i[5:0];        // 功能码
wire[4:0] op4 = inst_i[20:16];

```

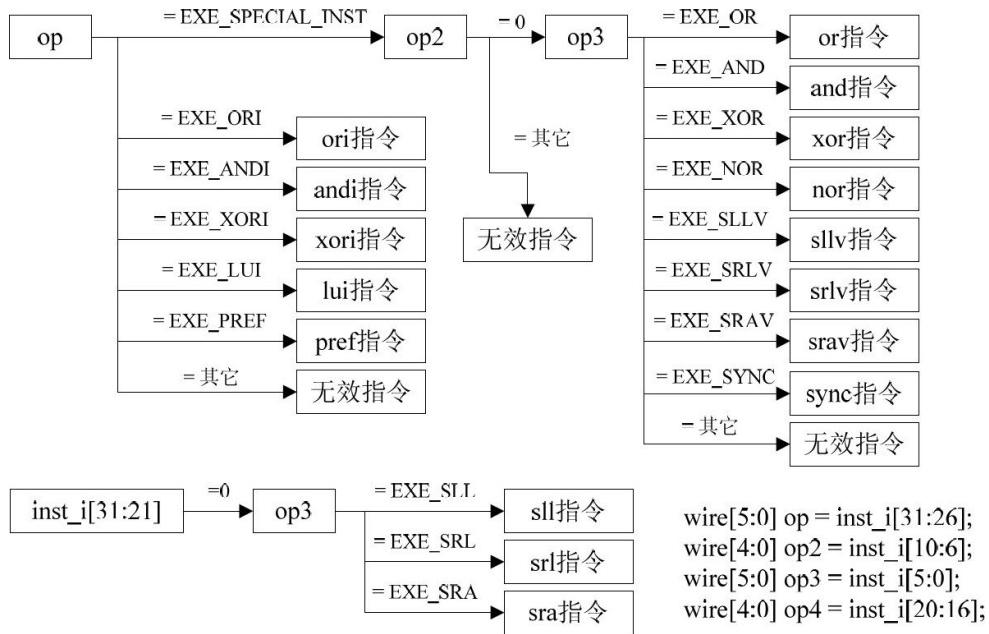


图5-15 确定指令种类的过程

首先依据指令码op进行判断，如果是SPECIAL类指令，再判断指令的第6~10bit（即op2）是否为0，如果为0，那么再依据功能码op3的值，进行最终判断，确定指令类型。如果指令码op不为SPECIAL，那么就直接依据指令码op的值进行判断。

只有在确定指令sll、srl、sra的时候有一点特殊，从图5-13可知，这3条指令都是SPECIAL类指令，但是这3条指令还要求第21~25bit为0，而且第6~10bit为移位位数，所以这3条指令的判断过程是：判断指令的第21~31bit是否全为0，如果全为0，那么再依据功能码op3进行最终判断，确定指令类型。

ID模块主要修改内容如下，完整的代码可以参考本书附带光盘中Code\Chapter5_2目录下的id.v文件。

```
module id(
```

```
....
```

```

);

wire[5:0] op  = inst_i[31:26];
wire[4:0] op2 = inst_i[10:6];
wire[5:0] op3 = inst_i[5:0];
wire[4:0] op4 = inst_i[20:16];
reg[`RegBus] imm;
reg instvalid;

always @ (*) begin
    if (rst == `RstEnable) begin
        aluop_o      <= `EXE_NOP_OP;
        alusel_o     <= `EXE_RES_NOP;
        wd_o         <= `NOPRegAddr;
        wreg_o       <= `WriteDisable;
        instvalid   <= `InstValid;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= `NOPRegAddr;
        reg2_addr_o <= `NOPRegAddr;
        imm          <= 32'h0;
    end else begin
        aluop_o      <= `EXE_NOP_OP;
        alusel_o     <= `EXE_RES_NOP;
        wd_o         <= inst_i[15:11];           //默认目的寄存器地
址wd_o
        wreg_o       <= `WriteDisable;
    end
end

```

```

instinvalid    <= `InstInvalid;
reg1_read_o   <= 1'b0;
reg2_read_o   <= 1'b0;
reg1_addr_o   <= inst_i[25:21];           // 默认的
reg1_addr_o
reg2_addr_o   <= inst_i[20:16];           // 默认的
reg2_addr_o
imm           <= `ZeroWord;
case (op)
`EXE_SPECIAL_INST:      begin
SPECIAL
case (op2)
5'b00000:             begin
case (op3)           //依据功能码判
断是哪种指令
`EXE_OR: begin          //or指令

wreg_o      <= `WriteEnable;
aluop_o     <= `EXE_OR_OP;
alusel_o    <= `EXE_RES_LOGIC;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instinvalid <= `InstValid;

```

```
end
```

```
`EXE_AND: begin //and指令
```

```
    wreg_o      <= `WriteEnable;  
    aluop_o     <= `EXE_AND_OP;  
    alusel_o    <= `EXE_RES_LOGIC;  
    reg1_read_o <= 1'b1;  
    reg2_read_o <= 1'b1;  
    instvalid   <= `InstValid;
```

```
end
```

```
`EXE_XOR: begin //xor指令
```

```
    wreg_o      <= `WriteEnable;  
    aluop_o     <= `EXE_XOR_OP;  
    alusel_o    <= `EXE_RES_LOGIC;  
    reg1_read_o <= 1'b1;  
    reg2_read_o <= 1'b1;  
    instvalid   <= `InstValid;
```

```
end
```

```
`EXE_NOR: begin          //nor指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_NOR_OP;
    alusel_o    <= `EXE_RES_LOGIC;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;

end
```

```
`EXE_SLLV: begin          //sllv指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_SLL_OP;
    alusel_o    <= `EXE_RES_SHIFT;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;

end
```

```
`EXE_SRLV: begin //srlv指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_SRL_OP;
    alusel_o    <= `EXE_RES_SHIFT;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;

end
```

```
`EXE_SRAV: begin //sraV指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_SRA_OP;
    alusel_o    <= `EXE_RES_SHIFT;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;

end
```

```
`EXE_SYNC: begin //sync指令
```

```
wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_NOP_OP;
alusel_o    <= `EXE_RES_NOP;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end
default: begin
end
endcase
end
default: begin
end
endcase
end

`EXE_ORI: begin //ori指令
```

```
wreg_o      <= `WriteEnable;
aluop_o     <= `EXE_OR_OP;
```

```
    alusel_o      <= `EXE_RES_LOGIC;  
    reg1_read_o  <= 1'b1;  
    reg2_read_o  <= 1'b0;  
    imm          <= {16'h0, inst_i[15:0]};  
    wd_o         <= inst_i[20:16];  
    instinvalid <= `InstInvalid;  
  
end
```

`*EXE_ANDI*: begin //andi指令

```
wreg_o      <= `WriteEnable;  
aluop_o      <= `EXE_AND_OP;  
alusel_o     <= `EXE_RES_LOGIC;  
reg1_read_o  <= 1'b1;  
reg2_read_o  <= 1'b0;  
imm          <= {16'h0, inst_i[15:0]};  
wd_o         <= inst_i[20:16];  
instinvalid <= `InstInvalid;  
  
end
```

`*EXE_XORI*: begin //xori指令

```
wreg_o      <= `WriteEnable;
aluop_o     <= `EXE_XOR_OP;
alusel_o    <= `EXE_RES_LOGIC;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
imm         <= {16'h0, inst_i[15:0]};
wd_o        <= inst_i[20:16];
instinvalid <= `InstInvalid;
end
```

`**EXE_LUI**: begin //lui指令

```
reg_o      <= `WriteEnable;
aluop_o     <= `EXE_OR_OP;
alusel_o    <= `EXE_RES_LOGIC;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
imm         <= {inst_i[15:0], 16'h0};
wd_o        <= inst_i[20:16];
instinvalid <= `InstInvalid;
end
```

`**EXE_PREF**: begin //pref指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_NOP_OP;
alusel_o     <= `EXE_RES_NOP;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b0;
instinvalid <= `InstInvalid;
end
default:      begin
end
endcase //case op

if (inst_i[31:21] == 11'b000000000000) begin

if (op3 == `EXE_SLL) begin          //sll指令

wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_SLL_OP;
alusel_o     <= `EXE_RES_SHIFT;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b1;
```

```
    imm[4:0]      <= inst_i[10:6];
    wd_o          <= inst_i[15:11];
    instinvalid   <= `InstInvalid;
```

end else if (op3 == `EXE_SRL) begin //srl指令

```
    wreg_o        <= `WriteEnable;
    aluop_o       <= `EXE_SRL_OP;
    alusel_o      <= `EXE_RES_SHIFT;
    reg1_read_o   <= 1'b0;
    reg2_read_o   <= 1'b1;
    imm[4:0]      <= inst_i[10:6];
    wd_o          <= inst_i[15:11];
    instinvalid   <= `InstInvalid;
```

end else if (op3 == `EXE_SRA) begin //sra指令

```
    wreg_o        <= `WriteEnable;
    aluop_o       <= `EXE_SRA_OP;
    alusel_o      <= `EXE_RES_SHIFT;
    reg1_read_o   <= 1'b0;
```

```

    reg2_read_o <= 1'b1;
    imm[4:0]      <= inst_i[10:6];
    wd_o          <= inst_i[15:11];
    instvalid    <= `InstValid;

end
end
end //if
end //always
.....
endmodule

```

对任一条指令而言，译码工作的主要内容是：确定要读取的寄存器情况、要执行的运算和要写入的目的寄存器三方面的信息。下面对其中几个典型指令的译码过程进行解释。

1. and指令的译码过程

and指令译码需要设置的三方面内容如下（or、xor、nor指令的译码过程可以参考and指令）。

(1) 要读取的寄存器情况：and指令需要读取rs、rt寄存器的值，所以设置reg1_read_o、reg2_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是and指令中的rs，默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是and指令中的rt。

(2) 要执行的运算：and指令要进行的是逻辑“与”操作，所以设置alusel_o为EXE_RES_LOGIC，设置aluop_o为EXE_AND_OP。

(3) 要写入的目的寄存器：and指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11～15bit，正是and指令中rd的位置。

2. andi指令的译码过程

andi指令译码需要设置的三方面内容如下（xori指令的译码过程可以参考andi指令）。

(1) 要读取的寄存器情况：andi指令只需要读取rs寄存器的值，所以设置reg1_read_o为1、reg2_read_o为0。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21～25bit，正是andi指令中的rs。设置reg2_read_o为0，暗含使用立即数作为运算的操作数。imm就是指令中的立即数进行零扩展后的值。

(2) 要执行的运算：andi指令要进行的是逻辑“与”操作，所以设置alusel_o为EXE_RES_LOGIC，设置aluop_o为EXE_AND_OP。这一点与and指令的译码过程一样。

(3) 要写入的目的寄存器：andi指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11～15bit，在此需要修改，对andi指令而言，目的寄存器地址是指令字的第16～20bit。

3. sllv指令的译码过程

sllv指令译码需要设置的三方面内容如下，（srlv、srav指令的译码过程可以参考sllv指令）。

(1) 要读取的寄存器情况：同and指令一样，设置reg1_read_o为1、reg2_read_o为1。

(2) 要执行的运算：sllv指令要进行的是逻辑左移操作，所以设置alusel_o为EXE_RES_SHIFT，设置aluop_o为EXE_SLL_OP。

(3) 要写入的目的寄存器：同and指令一样，设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11~15bit，正是sllv指令中rd的位置。

4. lui指令的译码过程

OpenMIPS将lui指令转化为ori指令来执行，语句如下。

```
lui rt,immediate = ori rt,$0,(immediate || 016)
)
```

也就是将指令中的立即数左移16bit，然后与\$0寄存器进行逻辑“或”运算。需要设置的三方面内容如下。

(1) 要读取的寄存器情况：需要读取寄存器\$0的值，所以设置reg1_read_o为1、reg2_read_o为0。默认通过Regfile模块读端口1读取的

寄存器地址reg1_addr_o的值是指令的第21~25bit，参考图5-10可知，正是0。设置imm为指令中的立即数左移16位的值。

(2) 要执行的运算：是逻辑“或”操作，所以alusel_o赋值为EXE_RES_LOGIC，aluop_o赋值为EXE_OR_OP。

(3) 要写入的目的寄存器：lui指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11~15bit，在此需要修改，对lui指令而言，目的寄存器地址是指令字的第16~20bit。

5. sll指令的译码过程

sll指令译码需要设置的三个方面内容如下（srl、sra指令的译码过程可以参考sll指令）。

(1) 要读取的寄存器情况：sll指令只需要读取rt寄存器的值，所以设置reg1_read_o为0、reg2_read_o为1。默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是sll指令中的rt。imm就是指令中的第6~10bit的值，参考图5-11可知，正是移位位数sa的值。

(2) 要执行的运算：sll指令要进行的是逻辑左移操作，所以设置alusel_o为EXE_RES_SHIFT，设置aluop_o为EXE_SLL_OP。

(3) 要写入的目的寄存器：sll指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，等于指令字的第11~15bit，正是sll指令中rd的位置。

5.5.2 修改执行阶段的EX模块

修改执行阶段EX模块的代码，主要修改内容如下，完整的代码可以参考本书光盘中Code\Chapter5_2目录下的ex.v文件。

```
module ex(
    .....
);

    reg[`RegBus] logicout;           // 保存逻辑运算结果
    reg[`RegBus] shiftres;          // 保存移位运算结果

    // 进行逻辑运算
    always @ (*) begin
        if(rst == `RstEnable) begin
            logicout <= `ZeroWord;
        end else begin
            case (aluop_i)
                `EXE_OR_OP:      begin      // 逻辑或运算
                    logicout <= reg1_i | reg2_i;
                end
                `EXE_AND_OP:    begin      // 逻辑与运算
                    logicout <= reg1_i & reg2_i;
                end
                `EXE_NOR_OP:   begin      // 逻辑或非运算
                    logicout <= ~(reg1_i | reg2_i);
                end
            endcase
        end
    end
);
```

```

        end

        `EXE_XOR_OP: begin          // 逻辑异或运算
            logicout <= reg1_i ^ reg2_i;
        end

        default: begin
            logicout <= `ZeroWord;
        end

    endcase

end      //if

end      //always

// 进行移位运算

always @ (*) begin
    if(rst == `RstEnable) begin
        shiftres <= `ZeroWord;
    end else begin
        case (aluop_i)
            `EXE_SLL_OP: begin          // 逻辑左移
                shiftres <= reg2_i << reg1_i[4:0] ;
            end
            `EXE_SRL_OP: begin          // 逻辑右移
                shiftres <= reg2_i >> reg1_i[4:0];
            end
            `EXE_SRA_OP: begin          // 算术右移
                shiftres <= ({32{reg2_i[31]}}<<(6'd32-
{1'b0,reg1_i[4:0]}))
                                | reg2_i >> reg1_i[4:0];
        end
    end

```

```
        end

        default: begin
            shiftres <= `ZeroWord;
        end

    endcase

end //if

end //always

// 依据alusel_i选择最终的运算结果
always @ (*) begin
    wd_o <= wd_i;
    wreg_o <= wreg_i;
    case ( alusel_i )
        `EXE_RES_LOGIC: begin
            wdata_o <= logicout;           // 选择逻辑运算结果为
最终运算结果
        end
        `EXE_RES_SHIFT: begin
            wdata_o <= shiftres;         // 选择移位运算结果为
最终运算结果
        end
        default: begin
            wdata_o <= `ZeroWord;
        end
    endcase
end
```

```
endmodule
```

上述代码主要是扩展了逻辑运算的过程，同时增加了进行移位运算的过程，最后，依据alusel_i的值，选择其中逻辑运算或移位运算的结果作为最终运算结果。

5.6 测试程序1——测试逻辑操作实现效果

编写如下测试程序用于检验逻辑操作指令是否正确实现，文件命名名为 inst_rom.S，在本附带光盘中 Code\Chapter5_2\AsmTest\LogicInstTest 目录下有测试程序的源文件。

```
.org 0x0
.global _start
.set noat
_start:
    lui $1,0x0101          # $1 = 0x01010000
    ori $1,$1,0x0101        # $1 = $1 | 0x0101 = 0x01010101
    ori $2,$1,0x1100        # $2 = $1 | 0x1100 = 0x01011101
    or   $1,$1,$2           # $1 = $1 | $2      = 0x01011101
    andi $3,$1,0x00fe       # $3 = $1 & 0x00fe = 0x00000000
    and  $1,$3,$1           # $1 = $3 & $1      = 0x00000000
    xorl $4,$1,0xff00       # $4 = $1 ^ 0xff00 = 0x0000ff00
```

```
xor $1,$4,$1          # $1 = $4 ^ $1      = 0x0000ff00  
nor $1,$4,$1          # $1 = $4 ~^ $1      = 0xfffff00ff
```

在程序的注释中给出了程序预期的执行效果，在这里就是寄存器\$1～\$4的变化情况。将上述inst_rom.S文件与第4章建立的Bin2Mem.exe、Makefile、ram.ld这三个文件复制到Ubuntu虚拟机中的同一个目录下，打开终端，使用cd命令进入该目录，然后输入make all，即可得到用于ModelSim仿真的inst_rom.data文件。

在ModelSim中新建一个工程，添加本书光盘中Code\Chapter5_2目录下的所有.v文件，然后可以编译。再将上面得到的inst_rom.data文件复制到ModelSim工程的目录下，就可以进行仿真了。上述仿真步骤以后不再重复说明。

ModelSim仿真结果如图5-16所示，regs[1]、regs[2]、regs[3]、regs[4]分别是寄存器\$1、\$2、\$3、\$4，观察这4个寄存器值的变化，可知符合预期，所以OpenMIPS正确实现了逻辑操作指令。

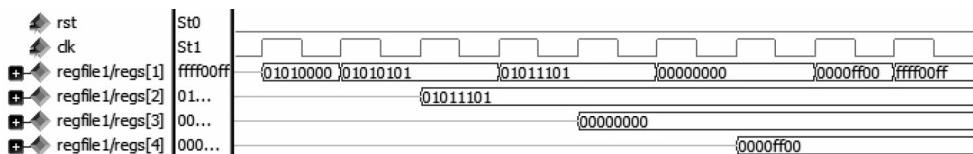


图5-16 逻辑操作指令测试例程仿真效果

5.7 测试程序2——测试移位操作与空指令实现效果

编写如下测试程序用于检验移位操作与空指令是否正确实现，文件名依然命名为inst_rom.S，在本书光盘中Code\Chapter5_2\AsmTest\ShiftInstTest目录下有测试程序的源文件。

```
.org 0x0
.set noat
.global _start
_start:
    lui    $2,0x0404      # $2 = 0x04040000
    ori    $2,$2,0x0404 # $2 = 0x04040000 | 0x0404 = 0x04040404
    ori    $7,$0,0x7
    ori    $5,$0,0x5
    ori    $8,$0,0x8
    sync
    sll    $2,$2,8       # $2 = 0x40404040 sll 8 = 0x04040400
    sllv   $2,$2,$7      # $2 = 0x04040400 sll 7 = 0x02020000
    srl    $2,$2,8       # $2 = 0x02020000 srl 8 = 0x00020200
    srlv   $2,$2,$5      # $2 = 0x00020200 srl 5 = 0x00001010
    nop
    pref
    sll    $2,$2,19     # $2 = 0x00001010 sll 19 = 0x80800000
    ssnop
    sra    $2,$2,16     # $2 = 0x80800000 sra 16 = 0xfffff8080
    sra   $2,$2,$8      # $2 = 0xfffff8080 sra 8 = 0xffffffff80
```

在程序的注释中给出了程序预期的执行效果，主要就是寄存器\$2的变化情况。ModelSim仿真结果如图5-17所示，观察寄存器\$2的变化

可以知道OpenMIPS正确实现了移位操作指令与空指令。

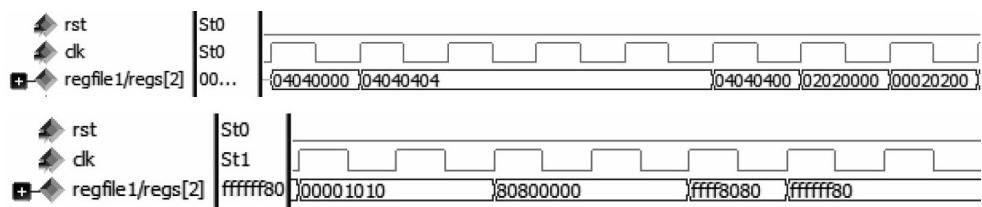


图5-17 移位操作指令与空指令的测试程序仿真结果

5.8 小结

本章首先分析了流水线中存在的数据相关问题，然后使用数据前推的方法解决了数据相关问题，随后修改OpenMIPS，实现了对逻辑、移位操作和空指令的支持，主要修改的是译码阶段的ID模块、执行阶段的EX模块。其中，在ID模块添加对新指令的译码，在EX模块添加对新的运算类型的支持。

第6章 移动操作指令的实现

本章将实现移动操作指令，首先在6.1节介绍了MIPS32指令集架构中定义的移动操作指令的格式、作用，接着在6.2节给出移动操作指令的实现思路，介绍了修改后的数据流图、新出现的数据相关问题及其解决措施，并给出了修改后的OpenMIPS系统结构图。在6.3节列出了详细的修改过程。本章最后通过一个测试程序验证移动操作指令是否正确实现。

6.1 移动操作指令说明

MIPS32指令集架构中定义的移动操作指令共有6条：movn、movz、mfhi、mthi、mflo、mtlo，后4条指令涉及对特殊寄存器HI、LO的读/写操作。截止到本章，我们的OpenMIPS处理器只实现了32个通用寄存器以及PC，所有的指令也只是对32个通用寄存器进行操作，还没有涉及特殊寄存器，本章将实现HI、LO这两个特殊寄存器。

HI、LO寄存器用于保存乘法、除法结果。当用于保存乘法结果时，HI寄存器保存结果的高32位，LO寄存器保存结果的低32位；当用于保存除法结果时，HI寄存器保存余数，LO寄存器保存商。在后续“算术操作指令的实现”一章中，会进一步说明。

这6条移动操作指令的格式如图6-1所示。

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000		rs	rt	rd	00000	MOVN 001011		movn指令
SPECIAL 000000		rs	rt	rd	00000	MOVZ 001010		movz指令
SPECIAL 000000	00000	00000		rd	00000	MFHI 010000		mfhi指令
SPECIAL 000000	00000	00000		rd	00000	MFLO 010010		mflo指令
SPECIAL 000000		rs	00000	00000	00000	MTHI 010001		mthi指令
SPECIAL 000000		rs	00000	00000	00000	MTLO 010011		mtlo指令

图6-1 移动操作指令的格式

从图6-1可知，这6条指令都是R类型指令，并且指令码都是6'b000000，即均为SPECIAL类指令；同时，指令第6~10bit都为0，可以依据指令中第0~5bit功能码的值判断是哪一种指令。各指令的用法及作用说明如下。

- 当功能码为6'b001011时，表示是movn指令。

指令用法为： movn rd, rs, rt。

指令作用为： if rt ≠ 0 then rd <- rs，判断地址为rt的通用寄存器的值。如果不为零，那么将地址为rs的通用寄存器的值赋给地址为rd的通用寄存器；反之，保持地址为rd的通用寄存器不变。movn是Move Conditional on Not Zero的意思。

- 当功能码为6'b001010时，表示是movz指令。

指令用法为： movz rd, rs, rt。

指令作用为： if $rt = 0$ then $rd \leftarrow rs$ ，与上面movn指令的作用正好相反，判断地址为 rt 的通用寄存器的值。如果为零，那么将地址为 rs 的通用寄存器的值赋给地址为 rd 的通用寄存器；反之，保持地址为 rd 的通用寄存器不变。movz是Move Conditional on Zero的意思。

- 当功能码为6'b010000时，表示是mfhi指令。

指令用法为： mfhi rd。

指令作用为： $rd \leftarrow hi$ ，将特殊寄存器HI的值赋给地址为 rd 的通用寄存器。

- 当功能码为6'b010010时，表示是mflo指令。

指令用法为： mflo rd。

指令作用为： $rd \leftarrow lo$ ，将特殊寄存器LO的值赋给地址为 rd 的通用寄存器。

- 当功能码为6'b010001时，表示是mthi指令。

指令用法为： mthi rs。

指令作用为： $hi \leftarrow rs$ ，将地址为 rs 的通用寄存器的值赋给特殊寄存器HI。

- 当功能码为6'b010011时，表示是mtlo指令。

指令用法为： mtlo rs。

指令作用为： $lo \leftarrow rs$ ，将地址为rs的通用寄存器的值赋给特殊寄存器LO。

6.2 移动操作指令实现思路

这6条移动操作指令可以分为两类：一类是不涉及特殊寄存器HI、LO的指令，包括movn、 movz；另一类是涉及特殊寄存器HI、 LO的指令，包括mfhi、 mflo、 mthi、 mtlo。前一类很好实现，基本思路与第5章实现逻辑、移位操作指令时类似，只需要修改ID、 EX模块即可。后一类涉及特殊寄存器HI、 LO，需要为OpenMIPS添加HI、 LO寄存器，以及相应的读/写控制。下面分别介绍各自的实现思路。

1. movn、 movz指令实现思路

与第5章逻辑、移位操作指令的实现过程类似。

(1) 在译码阶段给出运算类型alusel_o、运算子类型aluop_o、要写入的目的寄存器地址wd_o等信号的值，同时读取地址为rs、 rt的通用寄存器的值，但是这里需要新增一个步骤：依据读取地址为rt的通用寄存器的值是否为0，判断是否要写入目的寄存器。将上述结果送到执行阶段。

(2) 执行阶段依据传入的信号，确定最终要写入目的寄存器的信息（包含：是否写、写入的目的寄存器地址、写入的值），并将这些信息传递到访存阶段。

(3) 上述信息会一直传递到回写阶段。最后，依据这些信息修改目的寄存器，或者不做任何修改。

2. mthi、mtlo指令实现思路

这2条指令需要写HI、LO寄存器，与之前实现的通用寄存器一样，对HI、LO寄存器的写操作放在回写阶段进行。

(1) 在译码阶段依据指令，给出运算类型aluse_o、运算子类型aluop_o的值，同时读出地址为rs的通用寄存器的值。由于mthi、mtlo不写通用寄存器，所以wreg_o为WriteDisable，wd_o为0。

(2) 在执行阶段确定要写HI、LO寄存器的情况，以及要写入的值，并将这些信息传递到访存阶段。

(3) 访存阶段将这些信息再传递到回写阶段。

(4) 回写阶段依据这些信息修改HI、LO寄存器的值。

3. mfhi、mflo指令实现思路

这2条指令需要读HI、LO寄存器，设计在执行阶段才能读取到。

(1) 在译码阶段依据指令，给出运算类型aluse_o、运算子类型aluop_o的值，同时因为有要写入的目的寄存器，所以wreg_o为WriteEnable，wd_o为指令中rd的值，也就是目的寄存器地址。

(2) 在执行阶段获取HI或LO寄存器的值，作为要写入目的寄存器的数据，并将这些信息传递到访存阶段。

(3) 访存阶段将这些信息再传递到回写阶段。

(4) 回写阶段依据这些信息修改目的寄存器。

添加移动操作指令后的数据流图如图6-2所示。

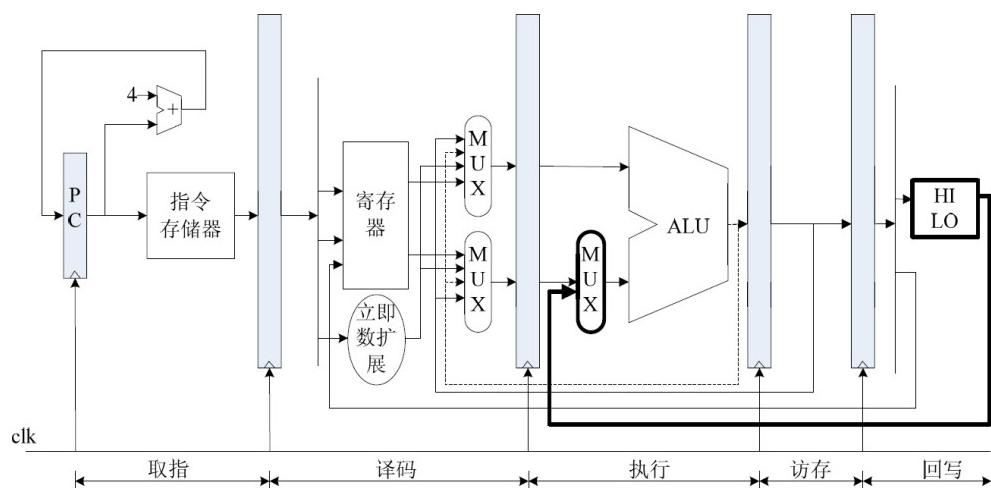


图6-2 添加移动操作指令后的数据流图

对比图6-2与图5-7可以发现有如下区别。

- 增加了HILO寄存器模块，并且该模块放在回写阶段。
- 将HI、LO寄存器的值传递到执行阶段，在执行阶段增加了一个选择模块，用于选择要参与运算的数据，如果是mfhi、mflo指令，那么就会选择传递过来的HI、LO寄存器的值。

6.2.1 新的数据相关情况的解决

进一步考虑mfhi、mflo指令的处理过程，这2条指令会在流水线执行阶段读取HI、LO寄存器的值，如果直接采用HILO模块给出的HI、LO寄存器的值，可能不是正确的HI、LO寄存器的值，因为此时处于访存、回写阶段的指令有可能会修改HI、LO寄存器，以如下程序为例。

```
1.    lui $1,0x0000          # $1 = 0x00000000
2.    lui $2,0xffff          # $1 = 0xffff0000
3.    mthi $0                # hi = 0x00000000
4.    mthi $1                # hi = 0x00000000
5.    mthi $2                # hi = 0xffff0000
6.    mfhi $4                # $4 = 0xffff0000
```

指令3、4、5均要修改HI寄存器，当指令6处于执行阶段时，指令5处于访存阶段，指令4处于回写阶段，而此时HI寄存器的值是指令3刚刚写入的0x00000000，HILO模块正是将该值传到执行阶段，如果采用这个值，那么就会出错，偏离程序设想，正确的值应该是当前处于访存阶段的指令5要写的数据，如图6-3所示。

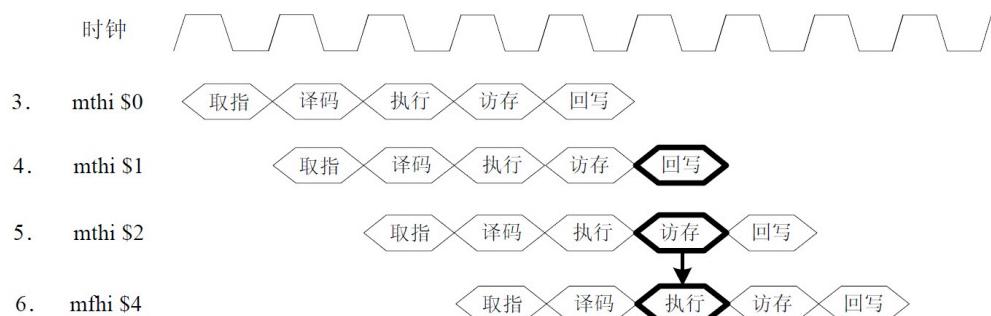


图6-3 HI、LO寄存器带来的数据相关示意图

似曾相识，是不是？这就是第5章介绍过的数据相关问题，解决措施还是使用数据前推。将处于访存阶段、回写阶段的指令对HI、LO寄

存器的操作信息反馈到执行阶段，执行阶段依据这些信息，确定HI、LO寄存器的正确值。

为此，需要修改数据流图如图6-4所示，相比图6-3，主要增加的部分就是将访存阶段、回写阶段的信息反馈到执行阶段，输入到执行阶段的选择模块（图中粗线所示），如果处于执行阶段的是mfhi、mflo指令，那么就会从中选择HI、LO寄存器的正确值。

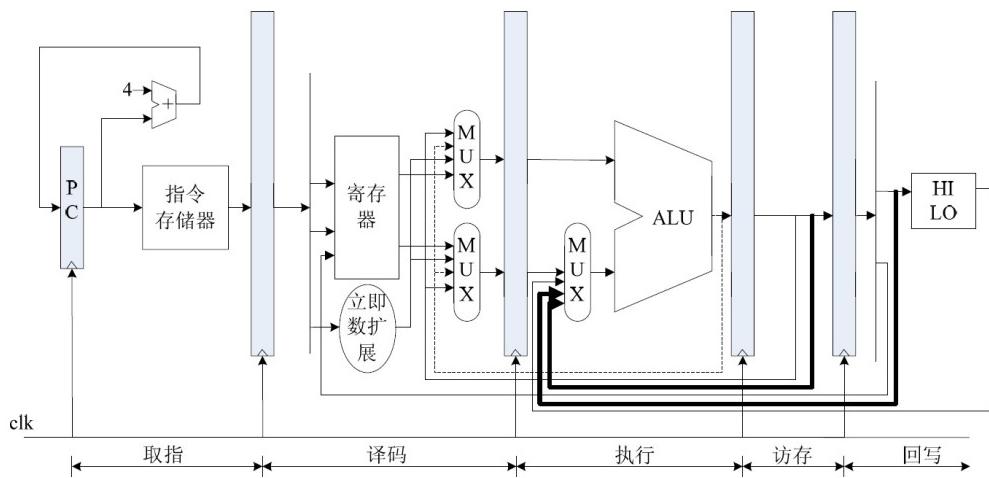


图6-4 解决HI、LO寄存器带来的数据相关问题后的数据流图

6.2.2 系统结构的修改

为了实现移动操作指令需要对OpenMIPS系统结构进行补充完善，主要修改如图6-5所示。

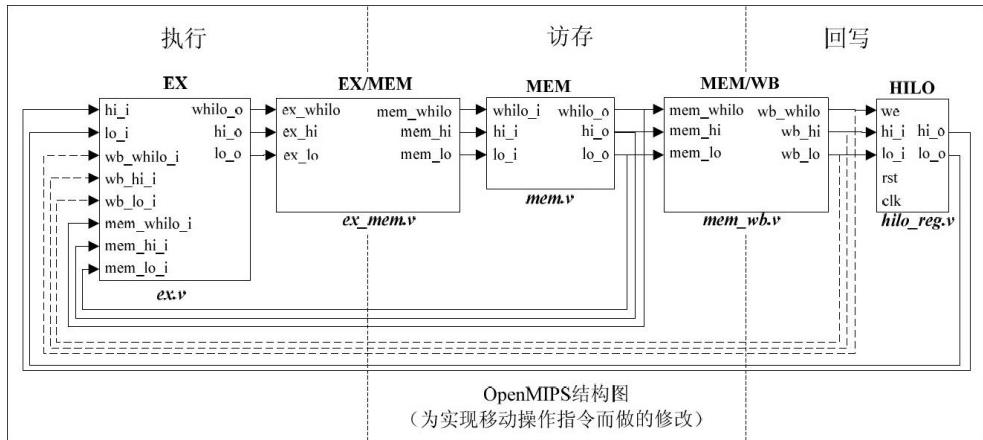


图6-5 为实现移动操作指令而对OpenMIPS系统结构所做的修改

主要有三个方面。

- (1) 增加了HILO模块，用于实现HI、LO寄存器。
- (2) 执行阶段的EX模块增加了whilo_o、hi_o、lo_o接口，分别表示是否要写HILO、要写入HI寄存器的值、要写入LO寄存器的值。这三个接口传递出来的对HI、LO寄存器的修改信息会通过EX/MEM、MEM、MEM/WB三个模块一直传递到回写阶段，并最终传递给HILO模块。
- (3) 执行阶段的EX模块增加了与HI、LO寄存器有关的输入接口，包括为解决HI、LO寄存器的数据相关问题而引入的接口，在6.3.3节会有详细介绍。

6.3 修改OpenMIPS以实现移动操作指令

6.3.1 HI、LO寄存器的实现

在HILO模块中实现HI、LO寄存器， HILO模块的接口描述如表6-1所示。

表6-1 HILO模块的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	we	1	输入	HI、LO 寄存器写使能信号
4	hi_i	32	输入	要写入 HI 寄存器的值
5	lo_i	32	输入	要写入 LO 寄存器的值
6	hi_o	32	输出	HI 寄存器的值
7	lo_o	32	输出	LO 寄存器的值

HILO模块的代码如下，源文件是本书附带光盘中Code\Chapter6目录下的hilo_reg.v。整个代码很简单：在时钟上升沿，如果复位信号无效，那么就判断输入的写使能信号we是否为WriteEnable，如果是WriteEnable，那么就将输入的hi_i、lo_i的值作为HI、LO寄存器的新值，并通过hi_o、lo_o接口输出。

```
module hilo_reg(  
  
    input wire                      clk,  
    input wire                      rst,  
  
    // 写端口  
    input wire                      we,  
    input wire[`RegBus]            hi_i,  
    input wire[`RegBus]            lo_i,  
  
    // 读端口
```

```
    output reg[`RegBus]    hi_o,
    output reg[`RegBus]    lo_o

);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        hi_o <= `ZeroWord;
        lo_o <= `ZeroWord;
    end else if((we == `WriteEnable)) begin
        hi_o <= hi_i;
        lo_o <= lo_i;
    end
end

endmodule
```

6.3.2 修改译码阶段的ID模块

在译码阶段要增加对移动操作指令的分析，根据图6-1给出的移动操作指令格式可知，这6条指令都是SPECIAL类指令，且第6~10bit均为0，需要依据第0~5bit的功能码确定指令，确定指令的过程如图6-6所示。

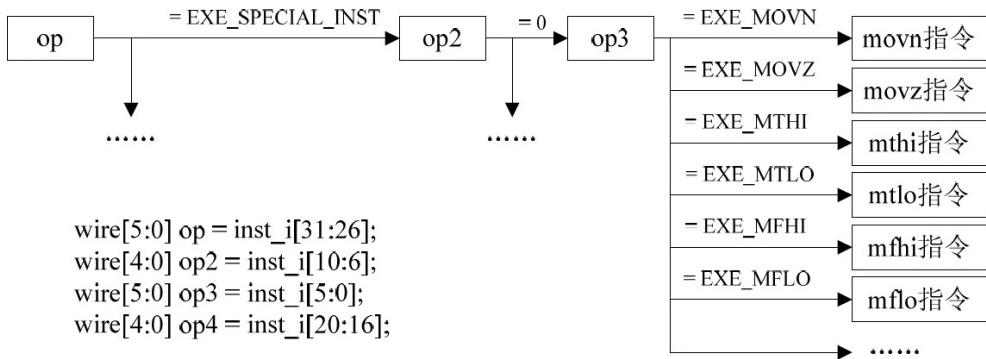


图6-6 确定移动操作指令的过程

其中涉及的宏定义如下，正是图6-1中各个指令的功能码。在本书附带光盘中Code\Chapter6目录下的defines.v文件中可以找到这些宏定义。

```

`define EXE_MOVZ 6'b001010
`define EXE_MOVN 6'b001011
`define EXE_MFHI 6'b010000
`define EXE_MTHI 6'b010001
`define EXE_MFL0 6'b010010
`define EXE_MTLO 6'b010011

```

译码阶段的ID模块主要修改如下。完整代码位于本书附带光盘中Code\Chapter6目录下的id.v文件。

```

module id(
    .....
);

    wire[5:0] op  = inst_i[31:26];
    wire[4:0] op2 = inst_i[10:6];
    wire[5:0] op3 = inst_i[5:0];

```

```

wire[4:0] op4 = inst_i[20:16];
.....
always @ (*) begin
    .....
    aluop_o      <= `EXE_NOP_OP;
    alusel_o     <= `EXE_RES_NOP;
    wd_o          <= inst_i[15:11];           // 默认目的寄存器地址
wd_o
    wreg_o       <= `WriteDisable;
    instinvalid <= `InstInvalid;
    reg1_read_o <= 1'b0;
    reg2_read_o <= 1'b0;
    reg1_addr_o <= inst_i[25:21];           // 默认的reg1_addr_o
    reg2_addr_o <= inst_i[20:16];           // 默认的reg2_addr_o
    imm          <= `ZeroWord;
    case (op)
        `EXE_SPECIAL_INST: begin           // 是SPECIAL类指
    令
        case (op2)
            5'b00000: begin                // op2为5'b00000
                case (op3)
                    .....
`EXE_MFHI: begin                   // mfhi指令

```

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_MFHI_OP;
alusel_o     <= `EXE_RES_MOVE;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b0;
instvalid   <= `InstValid;
end
```

`**EXE_MFL0**: begin // **mflo**指令

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_MFL0_OP;
alusel_o     <= `EXE_RES_MOVE;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b0;
instvalid   <= `InstValid;
end
```

`**EXE_MTHI**: begin // **mthi**指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_MTHI_OP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
instvalid   <= `InstValid;
end
```

`**EXE_MTLO:** begin // **mtlo**指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_MTLO_OP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
instvalid   <= `InstValid;
end
```

`**EXE_MOVN:** begin // **movn**指令

```
aluop_o      <= `EXE_MOVN_OP;
alusel_o     <= `EXE_RES_MOVE;
```

```
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid    <= `InstValid;
    //reg2_o的值就是地址为rt的通用寄存器的值
    if(reg2_o != `ZeroWord) begin
        wreg_o <= `WriteEnable;
    end else begin
        wreg_o <= `WriteDisable;
    end
end
```

`EXE_MOVZ: begin // movz指令

```
    aluop_o      <= `EXE_MOVZ_OP;
    alusel_o     <= `EXE_RES_MOVE;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid    <= `InstValid;
    //reg2_o的值就是地址为rt的通用寄存器的值
    if(reg2_o == `ZeroWord) begin
        wreg_o <= `WriteEnable;
    end else begin
        wreg_o <= `WriteDisable;
    end
```

```
    end  
    ....  
endmodule
```

有如下几点说明。

(1) 除mthi、 mtlo外的其余4条移动操作指令的运算类型alusel_o均为EXE_RES_MOVE。

(2) 指令mthi、 mtlo需要修改HI、 LO寄存器，但是不需要修改通用寄存器，所以在其译码结果中， wreg_o为WriteDisable。另外，设置reg1_read_o为1，表示需要通过Regfile模块读端口1读取通用寄存器的值，默认读取地址就是指令第21~25bit的值，正是mthi、 mtlo指令中的rs。读出的值作为要写入HI或LO寄存器的数据。

(3) movz指令的译码过程需要读取rs、 rt寄存器的值，所以设置reg1_read_o、 reg2_read_o均为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是movz指令中的rs， 默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是movz指令中的rt。所以， reg2_o的值就是读取到的地址为rt的寄存器的值，如果该值为0，那么设置wreg_o为WriteEnable， 表示要将地址为rs的寄存器的值赋给地址为rd的寄存器，反之， wreg_o为WriteDisable， 表示不赋值。

(4) movn指令的译码过程与 movz指令类似，只是wreg_o为WriteEnable的条件与movz正好相反。

6.3.3 修改执行阶段

1. 修改EX模块

译码阶段的结果会传递到执行阶段，执行阶段据此进行计算。考虑到执行阶段需要读写HI、LO寄存器，另外还要解决HI、LO寄存器带来的数据相关问题，所以需要给EX模块增加如表6-2所示的接口。各接口对外连接关系可以参考图6-5。

表6-2 EX模块要增加的接口

序号	接口名	宽度(bit)	输入/输出	作用
1	hi_i	32	输入	HILO模块给出的HI寄存器的值
2	lo_i	32	输入	HILO模块给出的LO寄存器的值
3	mem_whilo_i	1	输入	处于访存阶段的指令是否要写HI、LO寄存器
4	mem_hi_i	32	输入	处于访存阶段的指令要写入HI寄存器的值
5	mem_lo_i	32	输入	处于访存阶段的指令要写入LO寄存器的值
6	wb_whilo_i	1	输入	处于回写阶段的指令是否要写HI、LO寄存器
7	wb_hi_i	32	输入	处于回写阶段的指令要写入HI寄存器的值
8	wb_lo_i	32	输入	处于回写阶段的指令要写入LO寄存器的值
9	whilo_o	1	输出	执行阶段的指令是否要写HI、LO寄存器
10	hi_o	32	输出	执行阶段的指令要写入HI寄存器的值
11	lo_o	32	输出	执行阶段的指令要写入LO寄存器的值

EX模块的代码修改如下。完整代码位于本书附带光盘中Code\Chapter6目录下的ex.v文件中。

```
module ex(  
    . . . . .  
    // HILO模块给出的HI、LO寄存器的值  
    input wire[`RegBus]          hi_i,  
    input wire[`RegBus]          lo_i,
```

```
// 回写阶段的指令是否要写HI、LO，用于检测HI、LO寄存器带来的数据相关问题
    input wire[`RegBus]           wb_hi_i,
    input wire[`RegBus]           wb_lo_i,
    input wire                   wb_whilo_i,

// 访存阶段的指令是否要写HI、LO，用于检测HI、LO寄存器带来的数据相关问题
    input wire[`RegBus]           mem_hi_i,
    input wire[`RegBus]           mem_lo_i,
    input wire                   mem_whilo_i,

// 处于执行阶段的指令对HI、LO寄存器的写操作请求
    output reg[`RegBus]          hi_o,
    output reg[`RegBus]          lo_o,
    output reg                  whilo_o,
    .....
);

reg[`RegBus] logicout;           // 逻辑操作的结果
reg[`RegBus] shiftres;          // 移位操作的结果
reg[`RegBus] moveres;           // 移动操作的结果
reg[`RegBus] HI;                // 保存HI寄存器的最新值
reg[`RegBus] LO;                // 保存LO寄存器的最新值
```

```

      . . .

/*
*
** 第一段：得到最新的HI、LO寄存器的值，此处要解决数据相关问题 **
*/

```

always @ (*) begin

- if(rst == `RstEnable) begin
- {HI,LO} <= {`ZeroWord,`ZeroWord};
- end else if(mem_whilo_i == `WriteEnable) begin
- {HI,LO} <= {mem_hi_i,mem_lo_i}; // 访存阶段的指令要写
- HI、LO寄存器
- end else if(wb_whilo_i == `WriteEnable) begin
- {HI,LO} <= {wb_hi_i,wb_lo_i}; // 回写阶段的指令要写
- HI、LO寄存器
- end else begin
- {HI,LO} <= {hi_i,lo_i};
- end

end

```

/*
*
***** 第二段：MFHI、MFLO、MOVN、MOVZ指令 *****
*/

```

```
always @ (*) begin
    if(rst == `RstEnable) begin
        moveres <= `ZeroWord;
    end else begin
        moveres <= `ZeroWord;
        case (aluop_i)
            `EXE_MFHI_OP: begin
                // 如果是mfhi指令，那么将HI的值作为移动操作的结果
                moveres <= HI;
            end
            `EXE_MFL0_OP: begin
                // 如果是mflo指令，那么将LO的值作为移动操作的结果
                moveres <= LO;
            end
            `EXE_MOVZ_OP: begin
                // 如果是movz指令，那么将reg1_i的值作为移动操作的结果
                moveres <= reg1_i;
            end
            `EXE_MOVN_OP: begin
                // 如果是movn指令，那么将reg1_i的值作为移动操作的结果
                moveres <= reg1_i;
            end
            default : begin
                end
        endcase
    end
```

```
end

/*
*** 第三段：依据运算类型aluse1_i的值，确定wdata_o的值 ***
*/
always @ (*) begin
    wd_o    <= wd_i;
    wreg_o <= wreg_i;
    case ( aluse1_i )
        `EXE_RES_LOGIC: begin
            wdata_o <= logicout;
        end
        `EXE_RES_SHIFT: begin
            wdata_o <= shiftres;
        end
`EXE_RES_MOVE:      begin // 移动操作指令的aluse1_i为EXE_RES_MOVE
    wdata_o <= moveres;
```

end

```
default: begin
    wdata_o <= `ZeroWord;
end
endcase
end

//*****************************************************************************
/*
* 第四段：如果是MTHI、MTLO指令，那么需要给出whilo_o、hi_o、lo_i的值 *
*****
*/
always @ (*) begin
    if(rst == `RstEnable) begin
        whilo_o <= `WriteDisable;
        hi_o     <= `ZeroWord;
        lo_o     <= `ZeroWord;
    end else if(aluop_i == `EXE_MTHI_OP) begin
```

```

        whilo_o <= `WriteEnable;
        hi_o     <= reg1_i;
        lo_o     <= LO;           // 写HI寄存器，所以LO保持不变
    end else if(aluop_i == `EXE_MTL0_OP) begin
        whilo_o <= `WriteEnable;
        hi_o     <= HI;           // 写LO寄存器，所以HI保持不变
        lo_o     <= reg1_i;
    end else begin
        whilo_o <= `WriteDisable;
        hi_o     <= `ZeroWord;
        lo_o     <= `ZeroWord;
    end
end

endmodule

```

上面修改的代码可以分为四段理解。

(1) 第一段代码的作用是得到最新的HI、LO寄存器的值，首先判断当前处于访存阶段的指令是否要写HI、LO寄存器，即mem_whilo_o是否为WriteEnable，如果是，那么访存阶段的指令要写入的值就是HI、LO寄存器的最新值，如果不是，那么再判断当前处于回写阶段的指令是否要写HI、LO寄存器，如果是，那么回写阶段的指令要写入的值就是HI、LO寄存器的最新值，如果不是，那么从HILO模块输入的值hi_i、lo_i就是HI、LO寄存器的最新值。

(2) 第二段代码的作用是针对不同的移动操作指令，确定moveres的值，变量moveres存储的是移动操作指令的结果。

(3) 第三段代码的作用是依据运算类型alusel_i的值，将不同的运算结果赋给 wdata_o，如果是移动操作指令，那么 alusel_i 为 EXE_RES_MOVE，此时将moveres的值赋给wdata_o。

(4) 第四段代码的作用是确定是否要写HI、LO寄存器，如果是 mthi、mtlo 寄存器，那么要写 HI、LO 寄存器，所以设置输出信号 whilo_o 为 WriteEnable。具体地说，有如下两种情况。

- 如果是 mthi 指令，那么表示要写 HI 寄存器，所以 hi_o 等于 reg1_i 的值，参考译码阶段的 ID 模块可知，reg1_i 的值就是在译码阶段读出的地址为 rs 的寄存器的值。另外，LO 的值保持不变，所以 lo_o 等于 LO。
- 如果是 mtlo 指令，那么表示要写 LO 寄存器，所以 lo_o 等于 reg1_i 的值，参考译码阶段的 ID 模块可知，reg1_i 的值就是在译码阶段读出的地址为 rs 的寄存器的值。另外，HI 的值保持不变，所以 hi_o 等于 HI。

2. 修改EX/MEM模块

参考图6-5，EX模块新增加的输出接口whilo_o、hi_o、lo_o连接到 EX/MEM模块，需要给EX/MEM模块添加如表6-3所示的接口。

表6-3 EX/MEM模块要增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	ex_whilo	1	输入	执行阶段的指令是否要写 HI、LO 寄存器
2	ex_hi	32	输入	执行阶段的指令要写入 HI 寄存器的值
3	ex_lo	32	输入	执行阶段的指令要写入 LO 寄存器的值
4	mem_whilo	1	输出	访存阶段的指令是否要写 HI、LO 寄存器
5	mem_hi	32	输出	访存阶段的指令要写入 HI 寄存器的值
6	mem_lo	32	输出	访存阶段的指令要写入 LO 寄存器的值

EX/MEM模块的代码修改如下，完整代码位于本书附带光盘中Code\Chapter6目录下的ex_mem.v文件。主要修改的部分使用加粗、斜体表示，作用是将执行阶段得到的对HI、LO寄存器的写信息传递到访存阶段。

```
module ex_mem(  
  
    input wire      clk,  
    input  wire      rst,  
  
    // 来自执行阶段的信息  
    input wire[`RegAddrBus]      ex_wd,  
    input wire                  ex_wreg,  
    input wire[`RegBus]          ex_wdata,  
  
    input wire[`RegBus]          ex_hi,  
  
    input wire[`RegBus]          ex_lo,
```

```
input wire                                ex_whilo,  
  
// 送到访存阶段的信息  
output reg[`RegAddrBus]      mem_wd,  
output reg                      mem_wreg,  
output reg[`RegBus]          mem_wdata,  
  
output reg[`RegBus]           mem_hi,  
  
output reg[`RegBus]           mem_lo,  
  
output reg                     mem_whilo
```

```
);

always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        mem_wd      <= `NOPRegAddr;
        mem_wreg    <= `WriteDisable;
        mem_wdata   <= `ZeroWord;

mem_hi      <= `ZeroWord;

mem_lo      <= `ZeroWord;

mem_whilo <= `WriteDisable;
```

```
end else begin

    mem_wd      <= ex_wd;
    mem_wreg    <= ex_wreg;
    mem_wdata   <= ex_wdata;

mem_hi     <= ex_hi;

mem_lo     <= ex_lo;

mem_whilo <= ex_whilo;

    end
end

endmodule
```

6.3.4 修改访存阶段

1. 修改MEM模块

参考图6-5，EX/MEM模块新增加的输出接口mem_whilo、mem_hi、mem_lo连接到访存阶段的MEM模块，需要给MEM模块添加如表6-4所示的接口。

表6-4 MEM模块要增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	whilo_i	1	输入	访存阶段的指令是否要写 HI、LO 寄存器
2	hi_i	32	输入	访存阶段的指令要写入 HI 寄存器的值
3	lo_i	32	输入	访存阶段的指令要写入 LO 寄存器的值
4	whilo_o	1	输出	访存阶段的指令最终是否要写 HI、LO 寄存器
5	hi_o	32	输出	访存阶段的指令最终要写入 HI 寄存器的值
6	lo_o	32	输出	访存阶段的指令最终要写入 LO 寄存器的值

MEM模块的代码修改如下。对应本书附带光盘中Code\Chapter6目录下的mem.v文件。主要修改的部分使用加粗、斜体表示，作用是将对HI、LO寄存器的写信息传递到MEM/WB模块，后者会将这些信息传递到回写阶段。

```
module mem(  
  
    input wire          rst,  
  
    // 来自执行阶段的信息  
    input wire[`RegAddrBus]      wd_i,  
    input wire          wreg_i,
```

```
input wire[`RegBus]           wdata_i,  
  
input wire[`RegBus]           hi_i,  
  
input wire[`RegBus]           lo_i,  
  
input wire                   whilo_i,  
  
// 访存阶段的结果  
output reg[`RegAddrBus]       wd_o,  
output reg                   wreg_o,  
output reg[`RegBus]           wdata_o,  
  
output reg[`RegBus]           hi_o,
```

```
output reg[`RegBus]           lo_o,  
  
output reg                  whilo_o  
  
);  
  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        wd_o      <= `NOPRegAddr;  
        wreg_o   <= `WriteDisable;  
        wdata_o  <= `ZeroWord;  
  
        hi_o      <= `ZeroWord;
```

```
lo_o      <= `ZeroWord;

whilo_o <= `WriteDisable;

end else begin
    wd_o      <= wd_i;
    wreg_o   <= wreg_i;
    wdata_o <= wdata_i;

hi_o      <= hi_i;

lo_o      <= lo_i;
```

```
whilo_o <= whilo_i;
```

```
end
```

```
end
```

```
endmodule
```

2. 修改MEM/WB模块

参考图6-5， MEM模块新增的输出接口whilo_o、 hi_o、 lo_o连接到MEM/WB模块，需要给MEM/WB模块添加如表6-5所示的接口。

表6-5 MEM/WB模块要增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	mem_whilo	1	输入	访存阶段的指令是否要写 HI、 LO 寄存器
2	mem_hi	32	输入	访存阶段的指令要写入 HI 寄存器的值
3	mem_lo	32	输入	访存阶段的指令要写入 LO 寄存器的值
4	wb_whilo	1	输出	回写阶段的指令是否要写 HI、 LO 寄存器
5	wb_hi	32	输出	回写阶段的指令要写入 HI 寄存器的值
6	wb_lo	32	输出	回写阶段的指令要写入 LO 寄存器的值

MEM/WB 模块的代码修改如下。对应本书附带光盘中 Code\Chapter6 目录下的 mem_wb.v 文件。主要修改的部分使用加粗、斜体表示，作用是将对 HI、LO 寄存器的写信息传递到回写阶段。

```
module mem_wb(  
  
    input wire      clk,  
    input wire      rst,  
  
    // 访存阶段的结果  
    input wire[`RegAddrBus]      mem_wd,  
    input wire                  mem_wreg,  
    input wire[`RegBus]          mem_wdata,  
  
    input wire[`RegBus]          mem_hi,  
  
    input wire[`RegBus]          mem_lo,
```

```
input wire                                mem_whilo,  
  
// 送到回写阶段的信息  
output reg[`RegAddrBus]      wb_wd,  
output reg                      wb_wreg,  
output reg[`RegBus]          wb_wdata,  
  
output reg[`RegBus]           wb_hi,  
  
output reg[`RegBus]           wb_lo,  
  
output reg                     wb_whilo
```

```
);
```

```
    always @ (posedge clk) begin
        if(rst == `RstEnable) begin
            wb_wd      <= `NOPRegAddr;
            wb_wreg    <= `WriteDisable;
            wb_wdata   <= `ZeroWord;
```

```
        wb_hi      <= `ZeroWord;
```

```
        wb_lo      <= `ZeroWord;
```

```
        wb_whilo <= `WriteDisable;
```

```
    end else begin
        wb_wd      <= mem_wd;
        wb_wreg    <= mem_wreg;
        wb_wdata   <= mem_wdata;

wb_hi      <= mem_hi;

wb_lo      <= mem_lo;

wb_whilo  <= mem_whilo;

    end
end

endmodule
```

6.3.5 修改回写阶段

参考图6-5，MEM/WB模块输出的对HI、LO寄存器的写信息将直接送到HILO模块，包括：wb_whilo、wb_hi、wb_lo，后者据此修改HI、LO寄存器的值。

6.3.6 修改OpenMIPS顶层模块

由于本章增添了HILO模块，而且对流水线中的多个模块都增加了接口，所以需要修改OpenMIPS顶层模块，在其中将各个模块新增加的接口按照如图6-5所示的关系连接起来。因为很好理解，所以具体代码不在书中罗列，读者可以参考本书附带光盘中Code\Chapter6目录下的openmips.v文件。

6.4 测试程序

本节将通过一个测试程序验证为OpenMIPS处理器添加的移动操作指令是否实现正确，测试程序如下，对应本书附带光盘中Code\Chapter6\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.global _start
_start:
    // 给寄存器$1、$2、$3、$4赋初值
```

```
lui $1,0x0000          # $1 = 0x00000000
lui $2,0xffff          # $2 = 0xffff0000
lui $3,0x0505          # $3 = 0x05050000
lui $4,0x0000          # $4 = 0x00000000

// 对于movz指令而言，由于寄存器$1为0，所以将$2的值赋给$4
movz $4,$2,$1          # $4 = 0xffff0000

// 对于movn指令而言，由于寄存器$1为0，所以不赋值，$4保持不变
movn $4,$3,$1          # $4 = 0xffff0000

// 对于movn指令而言，由于寄存器$2不为0，所以将$3的值赋给$4
movn $4,$3,$2          # $4 = 0x05050000

// 对于movz指令而言，由于寄存器$3不为0，所以不赋值，$4的值保持不变
movz $4,$2,$3          # $4 = 0x05050000

// 连续三条mthi指令，分别将寄存器$0、$2、$3的值保存到HI寄存器
mthi $0                 # hi = 0x00000000
mthi $2                 # hi = 0xffff0000
mthi $3                 # hi = 0x05050000

// 读取HI寄存器的值到$4，同时可验证HI、LO寄存器带来的数据相关问题是否处理正确
mfhi $4                 # $4 = 0x05050000

// 连续三条指令mtlo，分别将寄存器$3、$2、$1的值保存到LO寄存器
```

```

mtlo $3          # lo = 0x05050000
mtlo $2          # lo = 0xfffff0000
mtlo $1          # lo = 0x00000000

// 读取LO寄存器的值到$4，同时可验证HI、LO寄存器带来的数据相关问题是否处理正确

mflo $4          # $4 = 0x00000000

```

程序的注释给出了预期效果，将上述inst_rom.S文件与第4章建立的Bin2Mem.exe、Makefile、ram.ld这三个文件复制到Ubuntu虚拟机中的同一个目录下，打开终端，使用cd命令进入该目录，然后输入make all，即可得到用于ModelSim仿真的指令存储器初始化文件inst_rom.data。

在ModelSim中新建一个工程，添加本书附带光盘中Code\Chapter6目录下的所有.v文件，然后可以编译。再将上面的inst_rom.data文件复制到ModelSim工程的目录下，就可以进行仿真了。

ModelSim仿真输出结果如图6-7、图6-8所示，观察\$4、HI、LO寄存器值的变化可以知道OpenMIPS正确实现了移动操作指令。

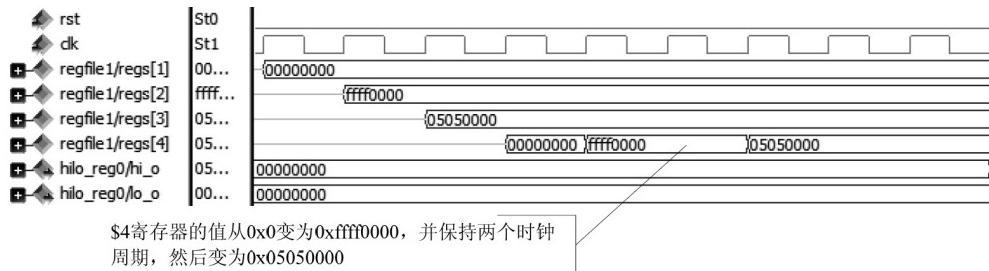


图6-7 移动操作指令仿真测试结果1

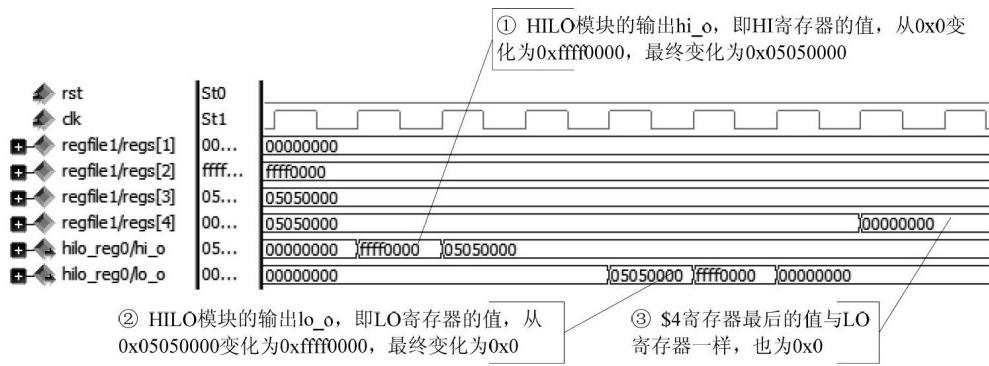


图6-8 移动操作指令仿真测试结果2

第7章 算术操作指令的实现

本章将实现MIPS32指令集架构定义的所有算术操作指令，共有21条，按照OpenMIPS实现这些指令的方式，可以分为三类，分别介绍如下。

(1) 简单算术操作指令

共有15条，包括加法、减法、比较、乘法等指令，这些指令在流水线的执行阶段都只需要一个时钟周期，而且实现思路很直观，与第4章添加逻辑操作指令类似，只需修改译码阶段的ID模块、执行阶段的EX模块，即可实现。

(2) 乘累加、乘累减指令

共有4条：乘累加（madd）、无符号乘累加（maddu）、乘累减（msub）、无符号乘累减（msubu）。其中madd、maddu要求操作数相乘后，再与HI、LO寄存器的值相加，msub、msubu指令要求操作数相乘后，再与HI、LO寄存器的值相减。也就是说，这4条指令都要做两次运算，一次乘法、一次加（减）法，如果将这两次运算放在流水线执行阶段的一个时钟周期中完成，那么会使流水线执行阶段所需要的时间明显增加，从而降低OpenMIPS工作时钟的频率，因此，OpenMIPS设计在流水线执行阶段使用两个时钟周期完成这类指令，一个时钟周期进行乘法，下一个时钟周期进行加（减）法。

(3) 除法指令

共有2条：有符号除法（div）、无符号除法（divu）。OpenMIPS计划采用试商法完成除法运算，对于32位的除法，流水线执行阶段至少需

要32个时钟周期。也就是说，除法指令需要多个时钟周期才能完成，所以单独作为一类。

本章将分别介绍上述三种类别的算术操作指令的实现过程。7.1节～7.4节给出了简单算术操作指令的格式和作用，介绍了实现思路，并修改OpenMIPS代码以实现简单算术操作指令，最后通过ModelSim仿真验证是否实现正确。

因为乘累加、乘累减、除法指令都需要在流水线执行阶段占用多个时钟周期，这就需要使流水线暂停，所以在实现这些指令之前，先要实现流水线暂停，在7.5节介绍了使流水线暂停的方法。

7.6节～7.9节给出了乘累加、乘累减指令的格式和作用，介绍了实现思路，并修改OpenMIPS代码以实现乘累加、乘累减指令，最后进行仿真测试。

7.10节～7.13节给出了除法指令的格式和作用，介绍了实现思路，并修改OpenMIPS代码以实现除法指令，最后进行仿真测试。

7.14节给出了实现算术操作指令后的数据流图。

7.1 简单算术操作指令说明

简单算术操作指令一共有15条，具体包括：add、addi、addiu、addu、sub、subu、clo、clz、slt、slti、sltiu、sltu、mul、mult、multu，各指令的格式及作用说明如下。

1. add、addu、sub、subu、slt、sltu指令

这6条指令的格式如图7-1所示，从图中可以发现这6条指令都是R类型指令，并且指令码都是6'b000000，即SPECIAL类。另外，第6~10bit都为0，需要依据指令中第0~5bit功能码的值进一步判断是哪一种指令。

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000		rs	rt	rd	00000	ADD 100000		add指令
SPECIAL 000000		rs	rt	rd	00000	ADDU 100001		addu指令
SPECIAL 000000		rs	rt	rd	00000	SUB 100010		sub指令
SPECIAL 000000		rs	rt	rd	00000	SUBU 100011		subu指令
SPECIAL 000000		rs	rt	rd	00000	SLT 101010		slt指令
SPECIAL 000000		rs	rt	rd	00000	SLTU 101011		sltu指令

图7-1 add、addu、sub、subu、slt、sltu指令格式

- 当功能码是6'b100000时，表示add指令，加法运算。

指令用法为：add rd, rs, rt。

指令作用为： $rd \leftarrow rs + rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行加法运算，结果保存到地址为rd的通用寄存器中。但是有一种特殊情况：如果加法运算溢出，那么会产生溢出异常，同时不保存结果。

- 当功能码是6'b100001时，表示addu指令，加法运算。

指令用法为：addu rd, rs, rt。

指令作用为： $rd \leftarrow rs + rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行加法运算，结果保存到地址为rd的通用寄存器中。

与add指令的不同之处在于addu指令不进行溢出检查，总是将结果保存到目的寄存器。

- 当功能码是6'b100010时，表示sub指令，减法运算。

指令用法为：sub rd, rs, rt。

指令作用为： $rd \leftarrow rs - rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行减法运算，结果保存到地址为rd的通用寄存器中。但是有一种特殊情况：如果减法运算溢出，那么产生溢出异常，同时不保存结果。

- 当功能码是6'b100011时，表示subu指令，减法运算。

指令用法为：subu rd, rs, rt。

指令作用为： $rd \leftarrow rs - rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行减法运算，结果保存到地址为rd的通用寄存器中。与sub指令的不同之处在于：subu指令不进行溢出检查，总是将结果保存到目的寄存器。

- 当功能码是6'b101010时，表示slt指令，比较运算。

指令用法为：slt rd, rs, rt。

指令作用为： $rd \leftarrow (rs < rt)$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值按照有符号数进行比较，如果前者小于后者，那么将1保存到地址为rd的通用寄存器中；反之，将0保存到地址为rd的通用寄存器中。

- 当功能码是6'b101011时，表示sltu指令，比较运算。

指令用法为：sltu rd, rs, rt。

指令作用为： $rd \leftarrow (rs < rt)$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值按照无符号数进行比较，如果前者小于后者，那么将1保存到地址为rd的通用寄存器中；反之，将0保存到地址为rd的通用寄存器中。

2. addi、addiu、slti、sltiu指令

这4条指令的格式如图7-2所示，从图中可以发现这4条指令都是I类型指令，能够依据指令中第26~31bit指令码的值判断是哪一种指令。

31	26 25	21 20	16 15	0	
ADDI 001000	rs	rt	immediate		addi指令
ADDIU 001001	rs	rt	immediate		addiu指令
SLTI 001010	rs	rt	immediate		slti指令
SLTIU 001011	rs	rt	immediate		sltiu指令

图7-2 addi、addiu、slti、sltiu指令格式

- 当指令码是6'b001000时，表示addi指令，加法运算。

指令用法为：addi rt, rs, immediate。

指令作用为： $rt \leftarrow rs + (\text{sign_extended})\text{immediate}$ ，将指令中的16位立即数进行符号扩展，与地址为rs的通用寄存器的值进行加法运算，结果保存到地址为rt的通用寄存器中。但是有一个特殊情况：如果加法运算溢出，那么产生溢出异常，同时不保存结果。

- 当指令码是6'b001001时，表示addiu指令，加法运算。

指令用法为： addiu rt, rs, immediate。

指令作用为： $rt \leftarrow rs + (\text{sign_extended})\text{immediate}$ ，将指令中的16位立即数进行符号扩展，与地址为rs的通用寄存器的值进行加法运算，结果保存到地址为rt的通用寄存器中。与addi指令的区别在于：addiu指令不进行溢出检查，总是将结果保存到目的寄存器。

- 当指令码是6'b001010时，表示slti指令，比较运算。

指令用法为： slti rt, rs, immediate。

指令作用为： $rt \leftarrow (rs < (\text{sign_extended})\text{immediate})$ ，将指令中的16位立即数进行符号扩展，与地址为rs的通用寄存器的值按照有符号数进行比较，如果前者大于后者，那么将1保存到地址为rt的通用寄存器中；反之，将0保存到地址为rt的通用寄存器中。

- 当指令码是6'b001011时，表示sltiu指令，比较运算。

指令用法为： sltiu rt, rs, immediate。

指令作用为： $rt \leftarrow (rs < (\text{sign_extended})\text{immediate})$ ，将指令中的16位立即数进行符号扩展，与地址为rs的通用寄存器的值按照无符号数进行比较，如果前者大于后者，那么将1保存到地址为rt的通用寄存器中；反之，将0保存到地址为rt的通用寄存器中。

3. clo、clz指令

这2条指令的格式如图7-3所示，从图中可以发现这2条指令都是R类型指令，并且指令码都是6'b011100，在MIPS32指令集架构中表示SPECIAL2类。另外，第6~10bit都为0，需要依据指令中第0~5bit功能码的值进一步判断是哪一种指令。

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL2 011100	rs	rt	rd	00000	CLZ 100000	clz指令
SPECIAL2 011100	rs	rt	rd	00000	CLO 100001	clo指令

图7-3 clo、clz指令格式

- 当功能码是6'b100000时，表示clz指令，计数运算。

指令用法为：clz rd, rs。

指令作用为：rd <- coun_leading_zeros rs，对地址为rs的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为“1”的位，将该位之前“0”的个数保存到地址为rd的通用寄存器中，如果地址为rs的通用寄存器的所有位都为0（即0x00000000），那么将32保存到地址为rd的通用寄存器中。

- 当功能码是6'b100001时，表示clo指令，计数运算。

指令用法为：clo rd, rs。

指令作用为：rd <- coun_leading_ones rs，对地址为rs的通用寄存器的值，从其最高位开始向最低位方向检查，直到遇到值为“0”的位，将该位之前“1”的个数保存到地址为rd的通用寄存器中，如果地址为rs的通用寄存器的所有位都为1（即0xFFFFFFFF），那么将32保存到地址为rd的通用寄存器中。

4. multu、mult、mul指令

这3条指令的格式如图7-4所示，可知这3条指令都是R类型指令，并且mul指令的指令码是SPECIAL2，mult和multu的指令码是SPECIAL。

31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL2 011100	rs	rt	rd	00000	MUL 000010		mul指令
SPECIAL 000000	rs	rt	00000	00000	MULT 011000		mult指令
SPECIAL 000000	rs	rt	00000	00000	MULTU 011001		multu指令

图7-4 mul、mult、multu指令格式

- 当指令码为SPECIAL2，功能码为6'b000010时，表示mul指令，乘法运算。

指令用法为：mul rd, rs, st。

指令作用为：rd <- rs × rt，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为有符号数相乘，乘法结果的低32bit保存到地址为rd的通用寄存器中。

- 当指令码为SPECIAL，功能码为6'b011000时，表示mult指令，乘法运算。

指令用法为：mult rs, st。

指令作用为：{hi, lo} <- rs × rt，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为有符号数相乘，乘法结果的低32bit保存到LO寄存器中，高32bit保存到HI寄存器中。

- 当指令码为SPECIAL，功能码为6'b011001时，表示multu指令，乘法运算。

指令用法为： multu rs, st。

指令作用为： $\{hi, lo\} \leftarrow rs \times rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为无符号数相乘，乘法结果的低32bit保存到LO寄存器中，高32bit保存到HI寄存器中。与mult指令的区别在于：multu指令执行中将操作数作为无符号数进行运算。

7.2 简单算术操作指令实现思路

虽然简单算术操作指令的数目比较多，有15条，但实现方式都是相似的，与前几章逻辑、移位操作指令的实现方式也很类似，不需要增加新的模块和新的接口，只需要修改流水线译码阶段的ID模块、执行阶段的EX模块即可。实现思路如下。

(1) 修改流水线译码阶段的ID模块，添加对上述简单算术操作指令的译码，给出运算类型aluSel_o、运算子类型aluOp_o、要写入的目的寄存器地址wd_o等信息；同时根据需要，读取地址为rs、rt的通用寄存器的值。

(2) 修改流水线执行阶段的EX模块，依据传入的信息进行运算，得到运算结果，确定最终要写入目的寄存器的信息（包含：是否写、写入的目的寄存器地址、写入的值），并将这些信息传递到访存阶段。

(3) 上述信息会一直传递到回写阶段，最后修改目的寄存器。

7.3 修改OpenMIPS以实现简单算术操作指令

7.3.1 修改译码阶段的ID模块

在译码阶段要增加对简单算术操作指令的分析，分析的前提是能判断出指令的种类，根据图7-1至图7-4可以给出如图7-5所示的确定指令种类的过程。

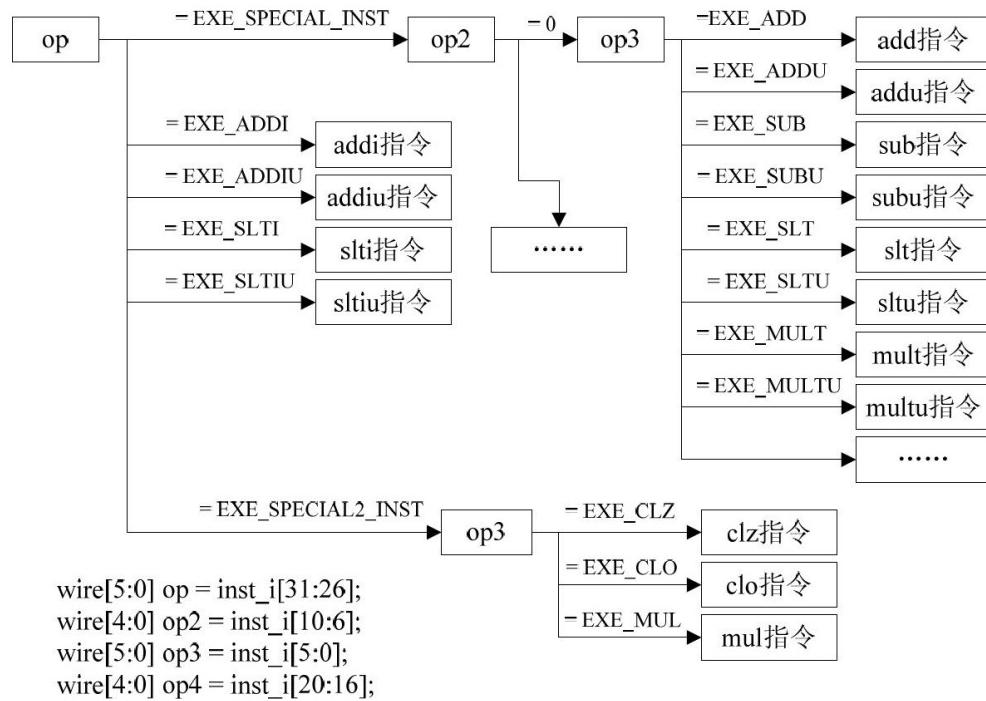


图7-5 确定简单算术操作指令的过程

其中涉及的宏定义如下，正是图7-5中各个指令的指令码或功能码。在本书附带光盘中Code\Chapter7_1目录下的defines.v文件中可以找到这些宏定义。

```
`define EXE_SLT    6'b101010
`define EXE_SLTU   6'b101011
`define EXE_SLTI   6'b001010
`define EXE_SLTIU  6'b001011
`define EXE_ADD    6'b100000
`define EXE_ADDU   6'b100001
`define EXE_SUB    6'b100010
`define EXE_SUBU   6'b100011
`define EXE_ADDI   6'b001000
`define EXE_ADDIU  6'b001001
`define EXE_CLZ    6'b100000
`define EXE_CL0    6'b100001

`define EXE_MULT   6'b011000
`define EXE_MULTU  6'b011001
`define EXE_MUL    6'b000010
.....
`define EXE_SPECIAL_INST 6'b000000
`define EXE_REGIMM_INST  6'b000001
`define EXE_SPECIAL2_INST 6'b011100
```

修改 ID 模块的代码如下，完整代码位于本书附带光盘中 Code\Chapter7_1 目录下的 id.v 文件。

```
module id(
    .....
);
    .....
```

```

always @ (*) begin
    if (rst == `RstEnable) begin
        .....
    end else begin
        aluop_o      <= `EXE_NOP_OP;
        alusel_o     <= `EXE_RES_NOP;
        wd_o         <= inst_i[15:11];           // 默认目的寄存器地址wd_o
        wreg_o       <= `WriteDisable;
        instinvalid <= `InstInvalid;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= inst_i[25:21];           // 默认的reg1_addr_o
        reg2_addr_o <= inst_i[20:16];           // 默认的reg2_addr_o
        imm          <= `ZeroWord;
        case (op)
            `EXE_SPECIAL_INST: begin           // op等于SPECIAL
                case (op2)
                    5'b00000: begin             // op2等于5'b00000
                        case (op3)
                            .....
            `EXE_SLT: begin                  // slt指令
                wreg_o       <= `WriteEnable;
                aluop_o     <= `EXE_SLT_OP;

```

```
    alusel_o      <= `EXE_RES_ARITHMETIC;  
    reg1_read_o  <= 1'b1;  
    reg2_read_o  <= 1'b1;  
    instvalid    <= `InstValid;  
end
```

`**EXE_SLTU**: begin // **sltu**指令

```
wreg_o      <= `WriteEnable;  
aluop_o      <= `EXE_SLTU_OP;  
alusel_o      <= `EXE_RES_ARITHMETIC;  
reg1_read_o  <= 1'b1;  
reg2_read_o  <= 1'b1;  
instvalid    <= `InstValid;
```

end

`**EXE_ADD**: begin // **add**指令

```
wreg_o      <= `WriteEnable;  
aluop_o      <= `EXE_ADD_OP;  
alusel_o      <= `EXE_RES_ARITHMETIC;  
reg1_read_o  <= 1'b1;
```

```
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;
end

`EXE_ADDU: begin                                // addu指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_ADDU_OP;
    alusel_o    <= `EXE_RES_ARITHMETIC;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;
end

`EXE_SUB: begin                                // sub指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_SUB_OP;
    alusel_o    <= `EXE_RES_ARITHMETIC;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;
```

```
        end

`EXE_SUBU: begin                                // subu指令

    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_SUBU_OP;
    alusel_o    <= `EXE_RES_ARITHMETIC;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;

end

`EXE_MULT: begin                                // mult指令

    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_MULT_OP;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;

end

`EXE_MULTU: begin                               // multu指令
```

```
wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_MULTU_OP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instinvalid <= `InstValid;
end
default: begin
end
endcase // end case op3
end
default: begin
end
endcase // end case op2
end
.....
`EXE_SLTI: begin // slti指令
```

```
wreg_o      <= `WriteEnable;
aluop_o     <= `EXE_SLT_OP;
alusel_o    <= `EXE_RES_ARITHMETIC;
```

```
    reg1_read_o <= 1'b1;  
    reg2_read_o <= 1'b0;  
    imm         <= {{16{inst_i[15]}}, inst_i[15:0]};  
    wd_o        <= inst_i[20:16];  
    instvalid   <= `InstValid;  
end
```

`EXE_SLTIU: begin // sltiu指令

```
wreg_o      <= `WriteEnable;  
aluop_o     <= `EXE_SLTU_OP;  
alusel_o    <= `EXE_RES_ARITHMETIC;  
reg1_read_o <= 1'b1;  
reg2_read_o <= 1'b0;  
imm         <= {{16{inst_i[15]}}, inst_i[15:0]};  
wd_o        <= inst_i[20:16];  
instvalid   <= `InstValid;  
end
```

`EXE_ADDI: begin // addi指令

```
wreg_o      <= `WriteEnable;
```

```
aluop_o      <= `EXE_ADDI_OP;
alusel_o     <= `EXE_RES_ARITHMETIC;
reg1_read_o  <= 1'b1;
reg2_read_o  <= 1'b0;
imm          <= {{16{inst_i[15]}}, inst_i[15:0]};
wd_o         <= inst_i[20:16];
instvalid    <= `InstValid;
end
```

`EXE_ADDIU: begin // addiu指令

```
wreg_o       <= `WriteEnable;
aluop_o      <= `EXE_ADDIU_OP;
alusel_o     <= `EXE_RES_ARITHMETIC;
reg1_read_o  <= 1'b1; reg2_read_o <= 1'b0;
imm          <= {{16{inst_i[15]}}, inst_i[15:0]};
wd_o         <= inst_i[20:16];
instvalid    <= `InstValid;
end
```

`EXE_SPECIAL2_INST: begin // op等于SPECIAL2

```
case ( op3 )
```

`EXE_CLZ: begin // clz指令

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_CLZ_OP;
alusel_o     <= `EXE_RES_ARITHMETIC;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
instinvalid <= `InstInvalid;
end
```

`*EXE_CL0*: begin // clo指令

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_CL0_OP;
alusel_o     <= `EXE_RES_ARITHMETIC;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
instinvalid <= `InstInvalid;
end
```

`*EXE_MUL*: begin // mul指令

```

        wreg_o      <= `WriteEnable;
        aluop_o     <= `EXE_MUL_OP;
        alusel_o    <= `EXE_RES_MUL;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b1;
        instvalid   <= `InstValid;
    end
    default: begin
    end
endcase //EXE_SPECIAL_INST2 case
end
default: begin
end
endcase //case op
.....
endmodule

```

对任一条指令而言，译码工作的主要内容是：确定要读取的寄存器情况、要执行的运算、要写的目的寄存器等三个方面的信息。下面对其中几个典型指令的译码过程进行解释。

1. add指令的译码过程

add指令译码需要设置的三方面内容如下（addu、sub、subu指令的译码过程可以参考add指令）。

(1) 要读取的寄存器情况：add指令需要读取rs、rt寄存器的值，所以设置reg1_read_o、reg2_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是add指令中的rs，默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是add指令中的rt所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。

(2) 要执行的运算：add指令是算术运算中的加法操作，所以此处将alusel_o 赋值为 EXE_RES_ARITHMETIC，aluop_o 赋值为 EXE_ADD_OP。

(3) 要写入的目的寄存器：add指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11~15bit，正是add指令中的rd。

2. addi指令的译码过程

addi指令译码需要设置的三方面内容如下（addiu、subi、subiu指令的译码过程可以参考addi指令）。

(1) 要读取的寄存器情况：addi指令只需要读取rs寄存器的值，所以设置reg1_read_o为1、reg2_read_o为0。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是addi指令中的rs。设置reg2_read_o为0，表示使用立即数作为参与运算的第二个操作数。imm就是指令中的立即数进行符号扩展后的值。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是imm的值。

(2) 要执行的运算：addi指令是算术运算中的加法操作，所以此处将alusel_o 赋值为 EXE_RES_ARITHMETIC，aluop_o 赋值为

EXE_ADDI_OP。

(3) 要写入的目的寄存器：addi指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置要写入的目的寄存器地址wd_o是指令中第16~20bit的值，正是addi指令中的rt。

3. slt指令的译码过程

slt指令译码需要设置的三方面内容如下（sltu指令的译码过程可以参考slt指令）。

(1) 要读取的寄存器情况：slt指令需要读取rs、rt寄存器的值，所以设置reg1_read_o、reg2_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是slt指令中的rs，默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是slt指令中的rt。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。

(2) 要执行的运算：slt指令是算术运算中的比较操作，所以此处将alusel_o赋值为EXE_RES_ARITHMETIC，aluop_o赋值为EXE_SLT_OP。

(3) 要写入的目的寄存器：slt指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令中第11~15bit的值，正是slt指令中的rd。

4. slti指令的译码过程

slti指令译码需要设置的三方面内容如下（sltiu指令的译码过程可以参考slti指令）。

(1) 要读取的寄存器情况：slti指令只需要读取rs寄存器的值，所以设置reg1_read_o为1、reg2_read_o为0。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是slti指令中的rs。设置reg2_read_o为0，表示使用立即数作为运算的第二个操作数。imm就是指令中的立即数进行符号扩展后的值。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是imm的值。

(2) 要执行的运算：slti指令是算术运算中的比较操作，所以此处将alusel_o赋值为EXE_RES_ARITHMETIC，aluop_o赋值为EXE_SLT_OP。

(3) 要写入的目的寄存器：slti指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置要写入的目的寄存器地址wd_o是指令中第16~20bit的值，正是slti指令中的rt。

5. mult指令的译码过程

mult指令译码需要设置的三方面内容如下（multu指令的译码过程可以参考mult指令）。

(1) 要读取的寄存器情况：mult指令需要读取rs、rt寄存器的值，所以设置reg1_read_o、reg2_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是mult指令中的rs，默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是mult指令中的rt。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。

(2) 要执行的运算：mult指令是乘法操作，并且乘法结果不需要写入通用寄存器，而是写入HI、LO寄存器，所以此处将alusel_o保持为默认值EXE_RES_NOP，aluop_o赋值为EXE_MULT_OP。

(3) 要写入的目的寄存器：mult指令不需要写通用寄存器，所以设置wreg_o为WriteDisable。

6. mul指令的译码过程

mul指令译码需要设置的三方面内容如下。

(1) 要读取的寄存器情况：mul指令需要读取rs、rt寄存器的值，所以设置reg1_read_o、reg2_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是mul指令中的rs；默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是mul指令中的rt。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。

(2) 要执行的运算：mul指令是乘法操作，并且乘法结果是写入通用寄存器，所以此处将alusel_o赋值为EXE_RES_MUL，aluop_o赋值为EXE_MUL_OP。

(3) 要写入的目的寄存器：mul指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11~15bit，正是mul指令中的rd。

7. clo指令的译码过程

clo指令译码需要设置的三方面内容如下（clz指令的译码过程可以参考clo指令）。

(1) 要读取的寄存器情况：clo指令只需要读取rs寄存器的值，所以设置reg1_read_o为1、reg2_read_o为0。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是clo指令中的rs，所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值。

(2) 要执行的运算：clo指令是算术运算中的计数操作，所以此处将alusel_o赋值为EXE_RES_ARITHMETIC，aluop_o赋值为EXE_CLO_OP。

(3) 要写入的目的寄存器：clo指令需要将结果写入目的寄存器，所以设置wreg_o为WriteEnable，设置wd_o为要写入的目的寄存器地址，默认是指令字的第11~15bit，正是clo指令中的rd。

7.3.2 修改执行阶段的EX模块

译码阶段的结果会传递到执行阶段，执行阶段的EX模块会据此进行运算，所以需要修改执行阶段EX模块的代码，主要修改内容如下，完整代码可以参考本书附带光盘中Code\Chapter7_1目录下的ex.v文件。

```
module ex(
    .....
);

    reg[`RegBus] logicout;
    reg[`RegBus] shiftres;
    reg[`RegBus] moveres;
```

```

reg[`RegBus] HI;
reg[`RegBus] LO;

// 新定义了一些变量

wire ov_sum; // 保存溢出情况
wire reg1_eq_reg2; // 第一个操作数是否等于第二个操
作数
wire reg1_lt_reg2; // 第一个操作数是否小于第二个操
作数
reg[`RegBus] arithmeticres; // 保存算术运算的结果
wire[`RegBus] reg2_i_mux; // 保存输入的第二个操作数
reg2_i的补码
wire[`RegBus] reg1_i_not; // 保存输入的第一个操作数
reg1_i取反后的值
wire[`RegBus] result_sum; // 保存加法结果
wire[`RegBus] opdata1_mult; // 乘法操作中的被乘数
wire[`RegBus] opdata2_mult; // 乘法操作中的乘数
wire[`DoubleRegBus] hilo_temp; // 临时保存乘法结果，宽度为64位
reg[`DoubleRegBus] mulres; // 保存乘法结果，宽度为64位

.....

```

***** 第一段：计算以下5个变量的值 *****

```

// (1) 如果是减法或者有符号比较运算，那么reg2_i_mux等于第二个操作数
//      reg2_i的补码，否则reg2_i_mux就等于第二个操作数reg2_i

```

```
assign reg2_i_mux = ((aluop_i == `EXE_SUB_OP) ||  
                      (aluop_i == `EXE_SUBU_OP) ||  
                      (aluop_i == `EXE_SLT_OP)) ?  
                        (~reg2_i)+1 : reg2_i;  
  
// (2) 分三种情况:  
//      A. 如果是加法运算, 此时reg2_i_mux就是第二个操作数reg2_i,  
//          所以result_sum就是加法运算的结果
```

```
//      B. 如果是减法运算，此时reg2_i_mux是第二个操作数reg2_i的补码，  
//          所以result_sum就是减法运算的结果  
//      C. 如果是有符号比较运算，此时reg2_i_mux也是第二个操作数reg2_i  
//          的补码，所以result_sum也是减法运算的结果，可以通过判断减法  
//          的结果是否小于零，进而判断第一个操作数reg1_i是否小于第二个操  
//          作数reg2_i
```

```
assign result_sum = reg1_i + reg2_i_mux;
```

```
// (3) 计算是否溢出，加法指令（add和addi）、减法指令（sub）执行的时候，
```

```
//      需要判断是否溢出，满足以下两种情况之一时，有溢出：  
//      A. reg1_i为正数，reg2_i_mux为正数，但是两者之和为负数  
//      B. reg1_i为负数，reg2_i_mux为负数，但是两者之和为正数
```

```
assign ov_sum = (!reg1_i[31] && !reg2_i_mux[31]) &&  
result_sum[31])
```

```
||((reg1_i[31] && reg2_i_mux[31]) && (!result_sum[31]));
```

// (4) 计算操作数1是否小于操作数2，分两种情况：

// A. aluop_i为EXE_SLT_OP表示有符号比较运算，此时又分3种情况

// A1. reg1_i为负数、reg2_i为正数，显然reg1_i小于reg2_i

// A2. reg1_i为正数、reg2_i为正数，并且reg1_i减去reg2_i的
值小于0

// (即result_sum为负)，此时也有reg1_i小于reg2_i

// A3. reg1_i为负数、reg2_i为负数，并且reg1_i减去reg2_i的
值小于0

// (即result_sum为负)，此时也有reg1_i小于reg2_i

// B、无符号数比较的时候，直接使用比较运算符比较reg1_i与reg2_i

```
assign reg1_lt_reg2 = ((aluop_i == `EXE_SLT_OP))?
```

```
((reg1_i[31] && !reg2_i[31]) ||
```

```
(!reg1_i[31] && !reg2_i[31] && result_sum[31])) ||  
  
(reg1_i[31] && reg2_i[31] && result_sum[31]))  
  
:(reg1_i < reg2_i);  
  
// (5) 对操作数1逐位取反，赋给reg1_i_not
```

```
assign reg1_i_not = ~reg1_i;
```

```

/*
***** 第二段：依据不同的算术运算类型，给arithmeticres变量赋值 ****
****

always @ (*) begin
    if(rst == `RstEnable) begin
        arithmeticres <= `ZeroWord;
    end else begin
        case (aluop_i)                                // aluop_i就
是运算类型
            `EXE_SLT_OP, `EXE_SLTU_OP: begin
                arithmeticres <= reg1_lt_reg2 ;      // 比较运算
            end
            `EXE_ADD_OP, `EXE_ADDU_OP, `EXE_ADDI_OP,
`EXE_ADDIU_OP:
            begin
                arithmeticres <= result_sum;        // 加法运算
            end
            `EXE_SUB_OP, `EXE_SUBU_OP: begin
                arithmeticres <= result_sum;        // 减法运算
            end
            `EXE_CLZ_OP: begin                   // 计数运算clz
                arithmeticres <= reg1_i[31] ? 0 : reg1_i[30] ?
1 :
                reg1_i[29] ? 2 : reg1_i[28]
? 3 :
                reg1_i[27] ? 4 : reg1_i[26]
? 5 :
                reg1_i[25] ? 6 : reg1_i[24]

```

```
? 7 :  
    reg1_i[23] ? 8 : reg1_i[22]  
? 9 :  
    reg1_i[21] ? 10 : reg1_i[20]  
? 11 :  
    reg1_i[19] ? 12 : reg1_i[18]  
? 13 :  
    reg1_i[17] ? 14 : reg1_i[16]  
? 15 :  
    reg1_i[15] ? 16 : reg1_i[14]  
? 17 :  
    reg1_i[13] ? 18 : reg1_i[12]  
? 19 :  
    reg1_i[11] ? 20 : reg1_i[10]  
? 21 :  
    reg1_i[9] ? 22 : reg1_i[8]  
? 23 :  
    reg1_i[7] ? 24 : reg1_i[6]  
? 25 :  
    reg1_i[5] ? 26 : reg1_i[4]  
? 27 :  
    reg1_i[3] ? 28 : reg1_i[2]  
? 29 :  
    reg1_i[1] ? 30 : reg1_i[0]  
? 31 : 32 ;  
end  
'EXE_CLO_OP: begin // 计数运算clo  
    arithmeticres <= (reg1_i_not[31] ? 0 :
```

```
reg1_i_not[29] ? 2 :  
reg1_i_not[28] ? 3 :  
reg1_i_not[27] ? 4 :  
reg1_i_not[26] ? 5 :  
reg1_i_not[25] ? 6 :  
reg1_i_not[24] ? 7 :  
reg1_i_not[23] ? 8 :  
reg1_i_not[22] ? 9 :  
reg1_i_not[21] ? 10 :  
reg1_i_not[20] ? 11 :  
reg1_i_not[19] ? 12 :  
reg1_i_not[18] ? 13 :  
reg1_i_not[17] ? 14 :  
reg1_i_not[16] ? 15 :  
reg1_i_not[15] ? 16 :  
reg1_i_not[14] ? 17 :  
reg1_i_not[13] ? 18 :  
reg1_i_not[12] ? 19 :  
reg1_i_not[11] ? 20 :  
reg1_i_not[10] ? 21 :  
reg1_i_not[9] ? 22 :  
reg1_i_not[8] ? 23 :  
reg1_i_not[7] ? 24 :  
reg1_i_not[6] ? 25 :  
reg1_i_not[5] ? 26 :  
reg1_i_not[4] ? 27 :  
reg1_i_not[3] ? 28 :  
reg1_i_not[2] ? 29 :
```

```

        reg1_i_not[1] ? 30 :
        reg1_i_not[0] ? 31 : 32) ;
    end
    default: begin
        arithmeticres <= `ZeroWord;
    end
endcase
end
end

// *****
***** 第三段：进行乘法运算 *****
***** */

// (1) 取得乘法运算的被乘数，如果是有符号乘法且被乘数是负数，那么取补码
assign opdata1_mult=(((aluop_i=='EXE_MUL_OP)||| (aluop_i=='EXE_MULT_OP))
&& (reg1_i[31] == 1'b1)) ? (~reg1_i +
1) : reg1_i;

// (2) 取得乘法运算的乘数，如果是有符号乘法且乘数是负数，那么取补码
assign opdata2_mult=(((aluop_i=='EXE_MUL_OP)||| (aluop_i=='EXE_MULT_OP))
&& (reg2_i[31] == 1'b1)) ? (~reg2_i +
1) : reg2_i;

// (3) 得到临时乘法结果，保存在变量hilo_temp中
assign hilo_temp = opdata1_mult * opdata2_mult;

```

// (4) 对临时乘法结果进行修正，最终的乘法结果保存在变量mulres中，主要有两点：

// A. 如果是有符号乘法指令mult、mul，那么需要修正临时乘法结果，如下：

// A1. 如果被乘数与乘数两者一正一负，那么需要对临时乘法结果

// hilo_temp求补码，作为最终的乘法结果，赋给变量mulres。

// A2. 如果被乘数与乘数同号，那么hilo_temp的值就作为最终的

// 乘法结果，赋给变量mulres。

// B. 如果是无符号乘法指令multu，那么hilo_temp的值就作为最终的乘法结果，

// 赋给变量mulres

always @ (*) begin

if(rst == `RstEnable) begin

mulres <= {`ZeroWord, `ZeroWord};

end else if ((aluop_i == `EXE_MULT_OP) || (aluop_i == `EXE_MUL_OP))

begin

if(reg1_i[31] ^ reg2_i[31] == 1'b1) begin

mulres <= ~hilo_temp + 1;

end else begin

mulres <= hilo_temp;

end

end else begin

mulres <= hilo_temp;

end

end

```

/*
*****第四段：确定要写入目的寄存器的数据*****
*/
always @ (*) begin
    wd_o <= wd_i;

    // 如果是add、addi、sub、subi指令，且发生溢出，那么设置wreg_o为
    // WriteDisable，表示不写目的寄存器
    if(((aluop_i == `EXE_ADD_OP) || (aluop_i == `EXE_ADDI_OP)
    ||
        (aluop_i == `EXE_SUB_OP)) && (ov_sum == 1'b1)) begin
        wreg_o <= `WriteDisable;
    end else begin
        wreg_o <= wreg_i;
    end

    case ( alusel_i )
        `EXE_RES_LOGIC:           begin
            wdata_o <= logicout;
        end
        `EXE_RES_SHIFT:   begin
            wdata_o <= shiftres;
        end
        `EXE_RES_MOVE:           begin
            wdata_o <= moveres;
        end
    end

```

```
`EXE_RES_ARITHMETIC: begin          //除乘法外的简单算术操作指令
wdata_o <= arithmeticres;

end

`EXE_RES_MUL:   begin          //乘法指令mul
wdata_o <= mulres[31:0];
```

```
end
```

```
    default:      begin
        wdata_o <= `ZeroWord;
    end
endcase
end
```

```
/*************************************************************************
***** 第五段：确定对HI、LO寄存器的操作信息 *****/

```

```
always @ (*) begin
    if(rst == `RstEnable) begin
        whilo_o <= `WriteDisable;
        hi_o <= `ZeroWord;
        lo_o <= `ZeroWord;
```

```
end else if((aluop_i == `EXE_MULT_OP) ||
```

```
(aluop_i == `EXE_MULTU_OP)) begin //mult、multu指令
    whilo_o <= `WriteEnable;
    hi_o <= mulres[63:32];
    lo_o <= mulres[31:0];
end else if(aluop_i == `EXE_MTHI_OP) begin
    whilo_o <= `WriteEnable;
```

```

    hi_o <= reg1_i;
    lo_o <= LO;
end else if(aluop_i == `EXE_MTL0_OP) begin
    whilo_o <= `WriteEnable;
    hi_o <= HI;
    lo_o <= reg1_i;
end else begin
    whilo_o <= `WriteDisable;
    hi_o <= `ZeroWord;
    lo_o <= `ZeroWord;
end
end

endmodule

```

上面的代码可以分为五段，大部分代码的含义都在注释中给出了详细解释，以下只做简要补充。

(1) 第一段代码计算出如下几个变量的值。

- reg2_i_mux：如果是减法或者有符号比较运算，那么reg2_i_mux等于第二个操作数reg2_i的补码，否则reg2_i_mux就等于第二个操作数reg2_i。
- result_sum：加、减法的结果。
- ov_sum：指示加、减法是否溢出。
- reg1_lt_reg2：操作数1是否小于操作数2。
- reg1_i_not：操作数1各位取反后的值。

(2) 第二段代码依据不同的算术运算类型，给变量arithmeticres赋值，此处只解释clz、clo指令的运算过程，其余指令的运算过程请参考程序注释。

- clz指令的作用是从最高位开始计数，直到遇到第一个1，所以在实现的时候就从最高位开始依次判断是否为1，如果为1，就给出当前已经数过的位数，如果没有为1的位，那么输出32。
- clo指令的作用是从最高位开始计数，直到遇到第一个0，效果等同于先将操作数取反，然后从最高位开始计数，直到遇到第一个1，所以在实现的时候就先对操作数取反，然后从最高位开始依次判断是否为1，如果为1，就给出当前已经数过的位数，如果没有为1的位，那么输出32。

(3) 第三段代码进行乘法运算。对于有符号乘法，要先求补码，再相乘，最后进行乘法结果的修正，乘法结果保存在变量mulres中。

(4) 第四段代码确定要写入目的寄存器的情况，有以下两点说明。

- 如果是add、addi、sub、subi指令，且发生溢出，那么设置wreg_o为WriteDisable，这样就不会写入目的寄存器。
- 如果是乘法指令以外的简单算术操作指令，那么将arithmeticres作为要写入目的寄存器的值。
- 如果是乘法指令mul，那么将乘法结果的低32位作为要写入目的寄存器的值。

(5) 第五段代码确定对HI、LO寄存器的写信息。如果是乘法指令mult、multu，那么需要写HI、LO寄存器，所以设置whilo_o为WriteEnable，写入HI寄存器的值为乘法结果的高32位，写入LO寄存器的值为乘法结果的低32位。

7.4 测试简单算术操作指令实现效果

本节通过实验来检验我们修改后的代码是否实现了简单算术操作指令，测试程序如下，源文件是本书附带光盘中Code\Chapter7_1\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.global _start
_start:

##### 第一段：测试add、addi、addiu、addu、sub、subu指令
#####

ori $1,$0,0x8000          # $1 = 0x00008000
sll $1,$1,16                # $1 = 0x80000000
ori $1,$1,0x0010          # $1 = 0x80000010    给$1赋初值

ori $2,$0,0x8000          # $2 = 0x00008000
sll $2,$2,16                # $2 = 0x80000000
ori $2,$2,0x0001          # $2 = 0x80000001    给$2赋初值

ori $3,$0,0x0000          # $3 = 0x00000000
addu $3,$2,$1              # $3 = 0x00000011    $1加$2，无符号加法
                            # $3 = 0x00000000
```

```

add $3,$2,$1          # $2加$1, 有符号加法, 结果溢出, 所以$3
应保持不变
                                # $3保持为0x00000000

sub $3,$1,$3          # $3 = 0x80000010 $1减去$3, 有符号
减法

subu $3,$3,$2         # $3 = 0xF           $3减去$2, 无符号
减法

addi $3,$3,2          # $3 = 0x11        $3加2, 有符号加法
ori $3,$0,0x0000       # $3 = 0x00000000
addiu $3,$3,0x8000     # $3 = 0xfffff8000 $3加0xfffff8000, 无符
号加法

#####
第二段：测试slt、sltu、slti、sltiu指令
#####

or $1,$0,0xffff      # $1 = 0x0000ffff
sll $1,$1,16          # $1 = 0xfffff0000 给$1赋初值
slt $2,$1,$0          # $2 = 1          比较$1与0x0, 有符号比较
sltu $2,$1,$0         # $2 = 0          比较$1与0x0, 无符号比较
slti $2,$1,0x8000     # $2 = 1          比较$1与0xfffff8000, 有符号
比较
sltiu $2,$1,0x8000    # $2 = 1          比较$1与0xfffff8000, 无符号
比较

#####
第三段：测试clo和clz指令
#####

```

```

lui $1, 0x0000          # $1 = 0x00000000 给$1赋初值
clo $2,$1               # $2 = 0x00000000 统计$1中“0”之前的“1”的
个数
clz $2,$1               # $2 = 0x00000020 统计$1中“1”之前的“0”的
个数

lui $1, 0xffff          # $1 = 0xfffff0000
ori $1,$1,0xffff        # $1 = 0xffffffff 给$1赋初值
clz $2,$1               # $2 = 0x00000000 统计$1中“1”之前的“0”的
个数
clo $2,$1               # $2 = 0x00000020 统计$1中“0”之前的“1”的
个数

lui $1, 0xa100          # $1 = 0xa1000000 给$1赋初值
clz $2,$1               # $2 = 0x00000000 统计$1中“1”之前的“0”的
个数
clo $2,$1               # $2 = 0x00000001 统计$1中“0”之前的“1”的
个数

lui $1, 0x1100          # $1 = 0x11000000 给$1赋初值
clz $2,$1               # $2 = 0x00000003 统计$1中“1”之前的“0”的
个数
clo $2,$1               # $2 = 0x00000000 统计$1中“0”之前的“1”的
个数

```


第四段：测试 mul、mult、multu 指令
#####

```

ori  $1,$0,0xffff
sll  $1,$1,16
ori  $1,$1,0xffffb      # $1 = -5    给$1赋初值
ori  $2,$0,6            # $2 = 6     给$2赋初值
mul  $3,$1,$2          # $3 = -30 = 0xffffffffe2
# $1乘以$2, 有符号乘法, 结果的低32位保存到$3

mult $1,$2             # HI = 0xffffffff
# LO = 0xffffffffe2
# $1乘以$2, 有符号乘法, 结果保存到HI、
LO寄存器

multu $1,$2            # HI = 0x5
# LO = 0xffffffffe2
# $1乘以$2, 无符号乘法, 结果保存到HI、
LO寄存器

nop
nop

```

程序的注释给出了预期效果，将上述inst_rom.S文件与第4章建立的Bin2Mem.exe、Makefile、ram.ld这三个文件复制到Ubuntu虚拟机中的同一个目录下，打开终端，使用cd命令进入该目录，然后输入make all，即可得到用于ModelSim仿真的指令存储器初始化文件inst_rom.data。

在ModelSim中新建一个工程，并添加本书附带光盘中Code\Chapter7_1目录下的所有.v文件，然后可以编译。再将上面的inst_rom.data文件复制到ModelSim工程的目录下，就可以进行仿真了。仿

真结果如图7-6、图7-7、图7-8、图7-9所示，分别对应测试程序中的四段。

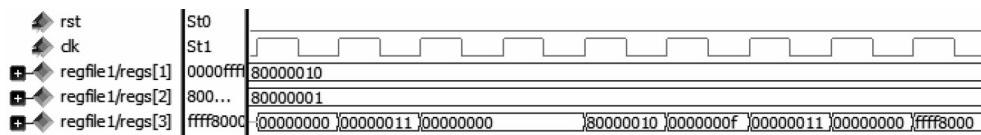


图7-6 观察寄存器\$3的变化，可知OpenMIPS正确实现了add、addi、addiu、addu、sub、subu指令，对应测试程序第一段

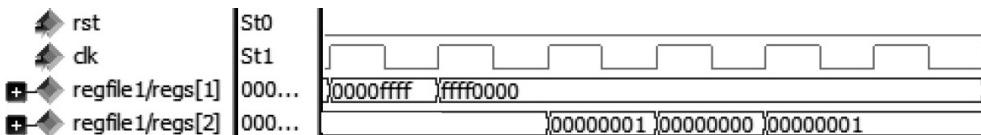


图7-7 观察寄存器\$2的变化，可知OpenMIPS正确实现了slt、sltu、slti、sltiu指令，对应测试程序第二段

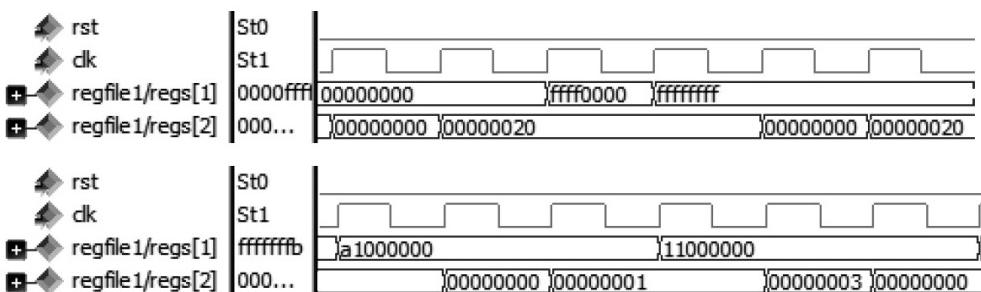


图7-8 观察寄存器\$2的变化，可知OpenMIPS正确实现了clz、clo指令，对应测试程序第三段

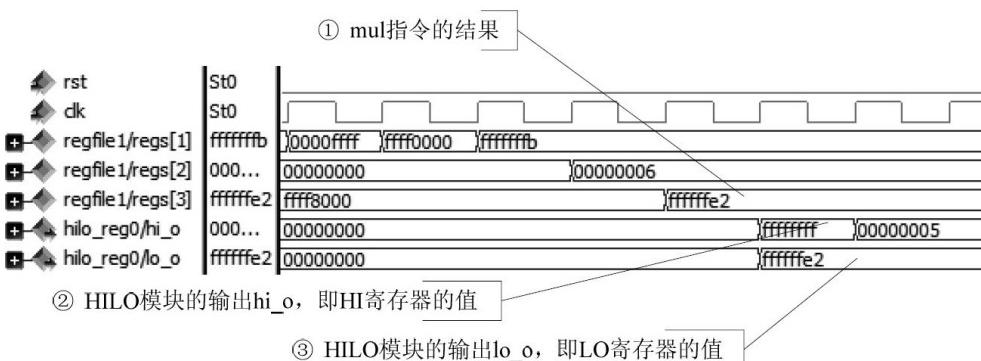


图7-9 观察寄存器\$3、HI、LO的变化，可知OpenMIPS正确实现了mul、mult、multu指令，对应测试程序第四段

7.5 流水线暂停机制的设计与实现

7.5.1 流水线暂停机制的设计

因为OpenMIPS设计乘累加、乘累减、除法指令在流水线执行阶段占用多个时钟周期，因此需要暂停流水线，以等待这些多周期指令执行完毕，一种直观的实现方法是：要暂停流水线，只需保持取指令地址PC的值不变，同时保持流水线各个阶段的寄存器（也就是IF/ID、ID/EX、EX/MEM、MEM/WB模块的输出）不变。

OpenMIPS采用的是一种改进的方法：假如位于流水线第n阶段的指令需要多个时钟周期，进而请求流水线暂停，那么需保持取指令地址PC的值不变，同时保持流水线第n阶段、第n阶段之前的各个阶段的寄存器不变，而第n阶段后面的指令继续运行。比如：流水线执行阶段的指令请求流水线暂停，那么保持PC不变，同时保持取指、译码、执行阶段的寄存器不变，但是可以允许访存、回写阶段的指令继续运行。

为此，设计添加CTRL模块，其作用是接收各阶段传递过来的流水线暂停请求信号，从而控制流水线各阶段的运行。

为了实现流水线暂停机制，对系统结构做如图7-10所示的修改。

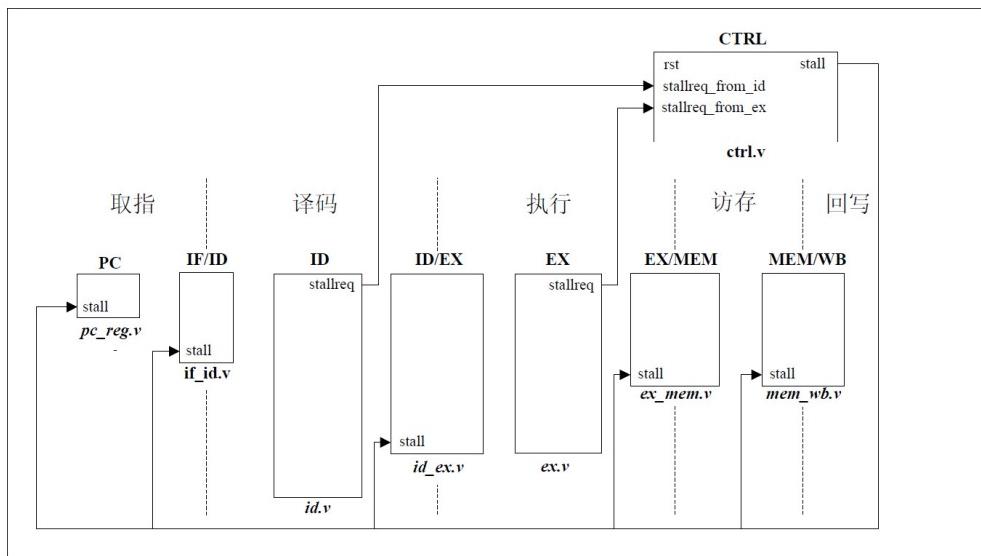


图7-10 为了实现流水线暂停机制而对系统结构所做的修改

CTRL模块的输入来自ID、EX模块的请求暂停信号stallreq，对于OpenMIPS教学版而言，只有译码、执行阶段可能会有暂停请求，取指、访存阶段都没有暂停请求，因为指令读取、数据存储器的读/写操作都可以在一个时钟周期内完成。

CTRL模块对暂停请求信号进行判断，然后输出流水线暂停信号stall。从图7-10中可知，stall输出到PC、IF/ID、ID/EX、EX/MEM、MEM/WB等模块，从而控制PC的值，以及流水线各个阶段的寄存器。

7.5.2 流水线暂停机制的实现

1. CTRL模块的实现

CTRL模块的接口如图7-10所示，各接口的作用如表7-1所示。

表7-1 CTRL模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	stallreq_from_id	1	输入	处于译码阶段的指令是否请求流水线暂停
3	stallreq_from_ex	1	输入	处于执行阶段的指令是否请求流水线暂停
4	stall	6	输出	暂停流水线控制信号

读者需要注意：输出信号stall是一个宽度为6的信号，其含义如下。

- stall[0]表示取指地址PC是否保持不变，为1表示保持不变。
- stall[1]表示流水线取指阶段是否暂停，为1表示暂停。
- stall[2]表示流水线译码阶段是否暂停，为1表示暂停。
- stall[3]表示流水线执行阶段是否暂停，为1表示暂停。
- stall[4]表示流水线访存阶段是否暂停，为1表示暂停。
- stall[5]表示流水线回写阶段是否暂停，为1表示暂停。

CTRL模块的代码如下，源文件是本书附带光盘中Code\Chapter7_2目录下的ctrl.v。

```
module ctrl(
    input wire    rst,
    input wire          stallreq_from_id,      // 来自译码阶段的暂停
    请求
    input wire          stallreq_from_ex,      // 来自执行阶段的暂停
    请求
    output reg[5:0]      stall
);

    always @ (*) begin
        if(rst == `RstEnable) begin

```

```
    stall <= 6'b0000000;

end else if(stallreq_from_ex == `Stop) begin

stall <= 6'b001111;

end else if(stallreq_from_id == `Stop) begin

stall <= 6'b000111;

    end else begin
        stall <= 6'b0000000;
    end
```

```
end  
  
endmodule
```

其中宏定义Stop在defines.v中定义如下：

```
`define Stop           1'b1          // 流水线暂停  
`define NoStop        1'b0          // 流水线继续
```

上述程序可以从以下三点理解。

(1) 当处于流水线执行阶段的指令请求暂停时（即stallreq_from_ex等于Stop），按照OpenMIPS流水线暂停机制的设计，要求取指、译码、执行阶段暂停，而访存、回写阶段继续，所以设置stall为6'b001111。

(2) 当处于流水线译码阶段的指令请求暂停时（即stallreq_from_id等于Stop），按照OpenMIPS流水线暂停机制的设计，要求取指、译码阶段暂停，而执行、访存、回写阶段继续，所以设置stall为6'b000111。

(3) 其余情况下，设置stall为6'b000000，表示不暂停流水线。

2. 修改取指阶段

(1) 修改PC模块

从图7-10中可知，PC模块新增加了一个输入信号stall，其值就是CTRL模块的接口stall。修改取指阶段PC模块的代码如下，其中修改的代码使用加粗、斜体标识。源文件是本书附带光盘中Code\Chapter7_2目录下的pc_reg.v文件。

```
module pc_reg(
    input wire                  clk,
    input wire                  rst,
    input wire[5:0]           stall, // 来自控制模块CTRL
    output reg[`InstAddrBus]   pc ,
    output reg                  ce
);

    always @ (posedge clk) begin
        if (rst == `RstEnable) begin
            ce <= `ChipDisable;
        end else begin
            ce <= `ChipEnable;
        end
    end

    // 当stall[0]为NoStop时, pc加4, 否则, 保持pc不变
    always @ (posedge clk) begin
        if (ce == `ChipDisable) begin
            pc <= 32'h00000000;
        end else
if(stall[0] == `NoStop)
```

```
begin
    pc <= pc + 4'h4;
end
end

endmodule
```

(2) 修改IF/ID模块

参考图7-10， IF/ID模块也新增了一个输入信号stall， 主要修改如下，
修改的代码使用加粗、 斜体标识。完整代码参考本书附带光盘中
Code\Chapter7_2目录下的if_id.v文件。

```
module if_id(
    .....
    input wire[5:0]          stall,
);
// (1) 当stall[1]为Stop, stall[2]为NoStop时, 表示取指阶段暂停,
//       而译码阶段继续, 所以使用空指令作为下一个周期进入译码阶段的指令
// (2) 当stall[1]为NoStop时, 取指阶段继续, 取得的指令进入译码阶段
// (3) 其余情况下, 保持译码阶段的寄存器id_pc、 id_inst不变
```

```
always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        id_pc    <= `ZeroWord;
        id_inst <= `ZeroWord;

end else if(stall[1] == `Stop && stall[2] == `NoStop) begin

    id_pc    <= `ZeroWord;

    id_inst <= `ZeroWord;

end else if(stall[1] == `NoStop) begin
```

```
id_pc    <= if_pc;  
  
id_inst <= if_inst;  
  
end  
  
end  
  
endmodule
```

3. 修改译码阶段

(1) 修改ID模块

参考图7-10，ID模块新增了一个输出信号stallreq，在实现加载、存储指令的时候会给该信号赋值，目前暂时设置这个输出就是NoStop，具体

代码不再给出，读者可以参考本书附带光盘中Code\Chapter7_2目录下的id.v文件。

(2) 修改ID/EX模块

参考图7-10，ID/EX模块新增了一个输入信号stall，主要修改如下，修改的代码使用加粗、斜体标识。完整代码位于本书附带光盘中Code\Chapter7_2目录下的id_ex.v文件。

```
module id_ex(  
    .....  
    //来自控制模块的信息  
  
    input wire[5:0]      stall,  
    .....  
);  
    // (1) 当stall[2]为Stop, stall[3]为NoStop时, 表示译码阶段暂停,  
    //       而执行阶段继续, 所以使用空指令作为下一个周期进入执行阶段的指令  
    // (2) 当stall[2]为NoStop时, 译码阶段继续, 译码后的指令进入执行阶段  
    // (3) 其余情况下, 保持执行阶段的寄存器ex_aluop、ex_alusel、  
    ex_reg1、  
    //      ex_reg2、ex_wd、ex_wreg不变  
    always @ (posedge clk) begin
```

```
if (rst == `RstEnable) begin
    . . .
end else if(stall[2] == `Stop && stall[3] == `NoStop) begin
    ex_aluop  <= `EXE_NOP_OP;
    ex_aluse1 <= `EXE_RES_NOP;
    ex_reg1   <= `ZeroWord;
    ex_reg2   <= `ZeroWord;
```

```
ex_wd      <= `NOPRegAddr;  
  
ex_wreg    <= `WriteDisable;  
  
end else if(stall[2] == `NoStop) begin  
  
    ex_aluop  <= id_aluop;
```

```
ex_aluse1 <= id_aluse1;
```

```
ex_reg1    <= id_reg1;
```

```
ex_reg2    <= id_reg2;
```

```
ex_wd      <= id_wd;
```

```
ex_wreg    <= id_wreg;
```

```
end  
  
end  
  
endmodule
```

4. 修改执行阶段

(1) 修改EX模块

参考图7-10， EX模块新增了一个输出信号stallreq_from_ex，在本章后面实现乘累加、乘累减、除法指令的时候会给该信号赋值。

(2) 修改EX/MEM模块

参考图7-10， EX/MEM模块新增了一个输入信号stall，主要修改如下，修改的代码使用加粗、斜体标识。完整代码位于本书附带光盘中Code\Chapter7_2目录下的ex_mem.v文件。

```
module ex_mem(  
    .....  
    //来自控制模块的信息
```

```
input wire[5:0]    stall,  
      .....  
);  
  // (1) 当stall[3]为Stop, stall[4]为NoStop时, 表示执行阶段暂停,  
  //       而访存阶段继续, 所以使用空指令作为下一个周期进入访存阶段的指令  
  // (2) 当stall[3]为NoStop时, 执行阶段继续, 执行后的指令进入访存阶段  
  // (3) 其余情况下, 保持访存阶段的寄存器mem_wb、mem_wreg、  
mwm_wdata,  
  //       mem_hi、mem_lo、mem_whilo不变  
  always @ (posedge clk) begin  
    if(rst == `RstEnable) begin  
      .....  
  
end else if(stall[3] == `Stop && stall[4] == `NoStop) begin  
  
mem_wd    <= `NOPRegAddr;
```

```
mem_wreg <= `WriteDisable;
```

```
mem_wdata <= `ZeroWord;
```

```
mem_hi      <= `ZeroWord;
```

```
mem_lo      <= `ZeroWord;
```

```
mem_whilo <= `WriteDisable;
```

```
end else if(stall[3] == `NoStop) begin
```

```
mem_wd      <= ex_wd;
```

```
mem_wreg    <= ex_wreg;
```

```
mem_wdata  <= ex_wdata;
```

```
mem_hi      <= ex_hi;  
  
mem_lo      <= ex_lo;  
  
mem_whilo  <= ex_whilo;  
  
      end      //if  
end        //always  
endmodule
```

5. 修改访存阶段

访存阶段只需要修改MEM/WB模块，参考图7-10，MEM/WB模块也新增了一个输入信号stall，主要修改如下，修改的代码使用加粗、斜体标

识。完整代码位于本书附带光盘中Code\Chapter7_2目录下的mem_wb.v文件。

```
module mem_wb(
    . . . .
    //来自控制模块的信息

    input wire[5:0]           stall,
    . . . .

);

    // (1) 当stall[4]为Stop, stall[5]为NoStop时, 表示访存阶段暂停,
    //       而回写阶段继续, 所以使用空指令作为下一个周期进入回写阶段的指令
    // (2) 当stall[4]为NoStop时, 访存阶段继续, 访存后的指令进入回写阶段
    // (3) 其余情况下, 保持回写阶段的寄存器wb_wd、wb_wreg、wb_wdata、
    //       wb_hi、wb_lo、wb_whilo不变
    always @ (posedge clk) begin
        if(rst == `RstEnable) begin
            . . . .
end else if(stall[4] == `Stop && stall[5] == `NoStop) begin
```

```
wb_wd      <= `NOPRegAddr;
```

```
wb_wreg   <= `WriteDisable;
```

```
wb_wdata <= `ZeroWord;
```

```
wb_hi      <= `ZeroWord;
```

```
wb_lo      <= `ZeroWord;
```

```
wb_whilo <= `WriteDisable;

end else if(stall[4] == `NoStop) begin

wb_wd      <= mem_wd;

wb_wreg   <= mem_wreg;
```

```
wb_wdata <= mem_wdata;  
  
wb_hi      <= mem_hi;  
  
wb_lo      <= mem_lo;  
  
wb_whilo <= mem_whilo;  
  
      end //if  
end      //always  
endmodule
```

6. 修改顶层模块OpenMIPS

因为上面添加了CTRL模块，而且对流水线各个阶段的模块也都增加了相应的接口，所以要修改OpenMIPS模块，以将新增接口与CTRL模块连接起来，连接关系如图7-10所示，具体的代码不在书中列出，读者可以参考本书附带光盘中Code\Chapter7_2目录下的openmips.v文件。

7.6 乘累加、乘累减指令说明

乘累加、乘累减指令共有4条，包括：madd、maddu、msub、msubu，各指令的格式如图7-11所示。从图中可知，这4条指令的指令码都是SPECIAL2，第6~15bit都为0，可以依据第0~5bit的功能码确定是哪一种指令。

31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL2 011100	rs	rt	00000	00000	MADD 000000		madd指令
SPECIAL2 011100	rs	rt	00000	00000	MADDU 000001		maddu指令
SPECIAL2 011100	rs	rt	00000	00000	MSUB 000100		msub指令
SPECIAL2 011100	rs	rt	00000	00000	MSUBU 000101		msubu指令

图7-11 madd、maddu、msub、msubu指令的格式

- 当功能码是6'b000000时，表示是madd指令，有符号乘累加运算。

指令用法为：madd rs, rt。

指令作用为： $\{HI, LO\} \leftarrow \{HI, LO\} + rs \times rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为有符号数进行乘法运算，运算结果与 $\{HI, LO\}$ 相加，相加的结果保存到 $\{HI, LO\}$ 中。此处 $\{HI, LO\}$ 表示HI、LO寄存器连接形成的64位数，HI是高32位，LO是低32位。

- 当功能码是6'b000001时，表示是maddu指令，无符号乘累加运算。

指令用法为：maddu rs, rt。

指令作用为： $\{HI, LO\} \leftarrow \{HI, LO\} + rs \times rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为无符号数进行乘法运算，运算结果与 $\{HI, LO\}$ 相加，相加的结果保存到 $\{HI, LO\}$ 中。

- 当功能码是6'b000100时，表示是msub指令，有符号乘累减运算。

指令用法为：msub rs, rt。

指令作用为： $\{HI, LO\} \leftarrow \{HI, LO\} - rs \times rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为有符号数进行乘法运算。然后使用 $\{HI, LO\}$ 减去乘法结果，相减的结果保存到 $\{HI, LO\}$ 中。

- 当功能码是6'b000101时，表示是msubu指令，无符号乘累减运算。

指令用法为：msubu rs, rt。

指令作用为： $\{HI, LO\} \leftarrow \{HI, LO\} - rs \times rt$ ，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值作为无符号数进行乘法运算。然后使用 $\{HI, LO\}$ 减去乘法结果，相减的结果保存到 $\{HI, LO\}$ 中。

7.7 乘累加、乘累减指令实现思路

在本章开始已经说明了乘累加、乘累减指令的实现思路，计划在流水线执行阶段采用两个时钟周期完成运算，第一个时钟周期进行乘法运算，第二个时钟周期将乘法结果与HI、LO寄存器进行加/减法。

为了实现乘累加、乘累减指令的实现思路，必须要保存两个信息：

(1) 当前是第几个时钟周期； (2) 乘法结果。OpenMIPS通过在EX/MEM模块中添加两个寄存器cnt、hilo，分别保存上述信息。修改系统结构如图7-12所示。

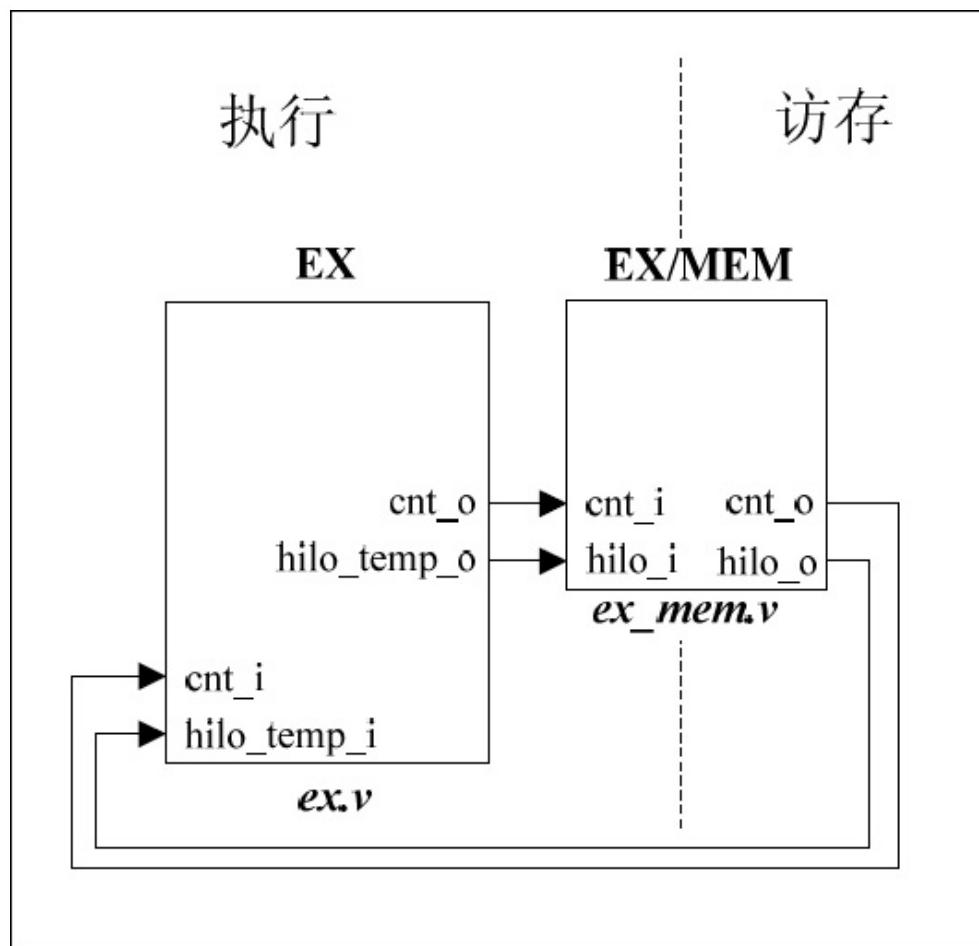


图7-12 为实现乘累加、乘累减指令而对系统结构做的修改

执行阶段EX模块的输出hilo_temp_o是乘法结果，传递到EX/MEM模块，并在下一个时钟周期送回EX模块，参与第二个时钟周期的加/减法运算。

执行阶段EX模块的输出cnt_o代表当前是第几个时钟周期，传递到EX/MEM模块，并在下一个时钟周期送回EX模块，后者据此判断当前处于乘累加、乘累减指令的第几个执行周期。

7.8 修改OpenMIPS以实现乘累加、乘累减指令

7.8.1 修改译码阶段的ID模块

译码阶段的ID模块要添加对乘累加、乘累减指令的分析，根据图7-11给出的指令格式可知，这4条指令都是SPECIAL2类指令，可以依据功能码确定是哪一种指令，确定指令的过程如图7-13所示。

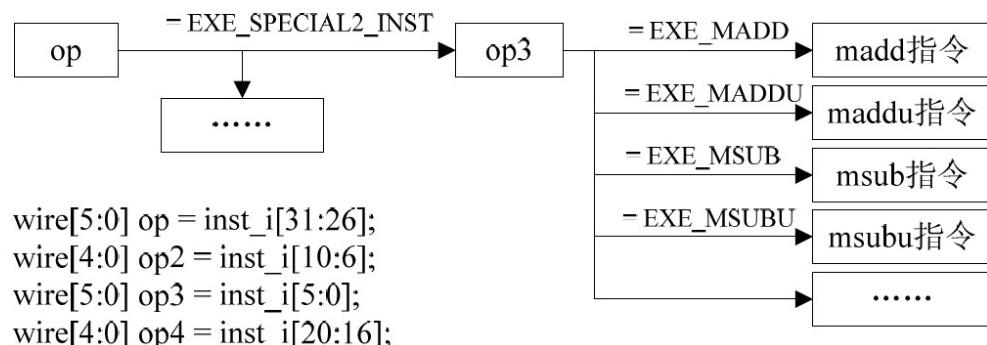


图7-13 确定乘累加、乘累减指令的过程

其中涉及的宏定义如下，正是图7-13中各个指令的功能码。在本书附带光盘中Code\Chapter7_2目录下的defines.v文件中可以找到这些定义。

```
`define EXE_MADD    6'b0000000
`define EXE_MADDU   6'b0000001
`define EXE_MSUB    6'b000100
`define EXE_MSUBU   6'b000101
```

译码阶段的ID模块主要修改内容如下，完整代码请参考本书光盘中Code\Chapter7_2目录下的id.v文件。

```
module id(
    .....
);

    .....
    assign stallreq = `NoStop;

    always @ (*) begin
        if (rst == `RstEnable) begin
            .....
        end else begin
            aluop_o      <= `EXE_NOP_OP;
            alusel_o     <= `EXE_RES_NOP;
            wd_o          <= inst_i[15:11];           // 默认目的寄存器地址
            wd_o
            wreg_o       <= `WriteDisable;
            instinvalid <= `InstInvalid;
            reg1_read_o  <= 1'b0;
            reg2_read_o  <= 1'b0;
            reg1_addr_o  <= inst_i[25:21];           // 默认的reg1_addr_o
```

```
reg2_addr_o <= inst_i[20:16];           // 默认的reg2_addr_o
imm          <= `ZeroWord;
case (op)
    .....
`EXE_SPECIAL2_INST: begin           // SPECIAL2类指令
    .....
    case ( op3 )
        .....
`EXE_MADD: begin           // madd指令
    .....
    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_MADD_OP;
    alusel_o    <= `EXE_RES_MUL;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid   <= `InstValid;
end
`EXE_MADDU: begin           // maddu指令
```

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_MADDU_OP;
alusel_o     <= `EXE_RES_MUL;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end
```

`**EXE_MSUB:** begin // *msub*指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_MSUB_OP;
alusel_o     <= `EXE_RES_MUL;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end
```

`**EXE_MSUBU:** begin // *msubu*指令

```

        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_MSUBU_OP;
        alusel_o    <= `EXE_RES_MUL;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b1;
        instvalid   <= `InstValid;
    end
    default: begin
    end
endcase //EXE_SPECIAL_INST2 case
.....
endmodule

```

这4条指令的译码过程都是相似的，简单说明如下。

(1) 因为最终结果都是写入HI、LO寄存器，而不是写入通用寄存器，所以设置wreg_o为WriteDisable。

(2) 因为都要读取两个寄存器的值，所以设置reg1_read_o、reg2_read_o为1'b1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o的值是指令的第21~25bit，正是指令中的rs；默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o的值是指令的第16~20bit，正是指令中的rt；所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。

(3) 运算类型alusel_o的值都设置为EXE_RES_MUL，不过由于没有要写的通用寄存器，所以此处alusel_o的值并没有作用，也可以设置为

EXE_RES_NOP。

(4) 运算子类型aluop_o的值设置为与具体的指令对应。

7.8.2 修改执行阶段的EX模块

参考图7-12可知，EX模块要增加4个接口，具体如表7-2所示。

表7-2 EX模块增加的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	hilo_temp_i	64	输入	第一个执行周期得到的乘法结果
2	cnt_i	2	输入	当前处于执行阶段的第几个时钟周期
3	hilo_temp_o	64	输出	第一个执行周期得到的乘法结果
4	cnt_o	2	输出	下一个时钟周期处于执行阶段的第几个时钟周期

EX模块的代码主要修改如下，完整代码请参考本书附带光盘中Code\Chapter7_2目录下的ex.v文件。

```
module ex(  
    . . . . .  
    // 增加的输入接口  
  
    input wire[`DoubleRegBus]      hilo_temp_i,  
    . . . . .
```

```
input wire[1:0]          cnt_i,  
.....  
// 增加的输出接口  
  
output reg[`DoubleRegBus]    hilo_temp_o,  
  
output reg[1:0]          cnt_o,  
  
output reg                stallreq  
);  
.....  
wire[`RegBus]          opdata1_mult;
```

```

    wire[`RegBus]          opdata2_mult;
    wire[`DoubleRegBus]   hilo_temp;
    reg[`DoubleRegBus]    hilo_temp1;
    reg                   stallreq_for_madd_ms sub;

    . . .

/*
***** 第一段：计算乘法结果 *****
*/
// (1) 取得乘法操作的被乘数，指令madd、msub都是有符号乘法，如果第一个
//      操作数reg1_i是负数，那么取reg1_i的补码作为被乘数；反之，直接
//      使用reg1_i作为被乘数
assign opdata1_mult = (((aluop_i == `EXE_MUL_OP) ||
                        (aluop_i == `EXE_MULT_OP) ||
                        (aluop_i == `EXE_MADD_OP) ||
                        (aluop_i == `EXE_MSUB_OP))
&&                                         (reg1_i[31] == 1'b1)) ?
                        (~reg1_i + 1) : reg1_i;

// (2) 取得乘法操作的乘数，指令madd、msub是有符号乘法，如果第二个
//      操作数reg2_i是负数，那么取reg2_i的补码作为乘数；反之，直接

```

```

//      使用reg2_i作为乘数

assign opdata2_mult = (((aluop_i == `EXE_MUL_OP) ||
                      (aluop_i == `EXE_MULT_OP) ||
                      (aluop_i == `EXE_MADD_OP) ||
                      (aluop_i == `EXE_MSUB_OP))

                                          && (reg2_i[31] == 1'b1)) ?
                      (~reg2_i + 1) : reg2_i;

// (3) 得到临时乘法结果，保存在变量hilo_temp中
assign hilo_temp = opdata1_mult * opdata2_mult;

// (4) 对临时乘法结果进行修正，最终的乘法结果保存在变量mulres中，有
两种情况：
//      A. 如果是有符号乘法运算madd、msub，那么需要修正临时乘法结
果，如下：
//          A1. 如果被乘数与乘数两者一正一负，那么需要对临时乘法结果
//                  hilo_temp求补码，作为最终的乘法结果，赋给变
//      mulres。
//          A2. 如果被乘数与乘数同号，那么hilo_temp的值就作为mulres
//                  的值。
//      B. 如果是无符号乘法运算maddu、msubu，那么hilo_temp的值就作
为
//          最终的乘法结果，赋给变量mulres
always @ (*) begin

```

```

if(rst == `RstEnable) begin
    mulres <= {`ZeroWord, `ZeroWord};
end else if ((aluop_i == `EXE_MULT_OP) ||
             (aluop_i == `EXE_MUL_OP) ||
             (aluop_i == `EXE_MADD_OP) ||
             (aluop_i == `EXE_MSUB_OP))

begin
    if(reg1_i[31] ^ reg2_i[31] == 1'b1) begin
        mulres <= ~hilo_temp + 1;
    end else begin
        mulres <= hilo_temp;
    end
end else begin
    mulres <= hilo_temp;
end

end

/****************************************
***** 第二段：乘累加、乘累减 *****
****************************************/
// MADD、MADDU、MSUB、MSUBU指令

always @ (*) begin

```

```
if(rst == `RstEnable) begin
    hilo_temp_o <= {`ZeroWord, `ZeroWord};
    cnt_o       <= 2'b00;
    stallreq_for_madd_msub <= `NoStop;
end else begin
    case (aluop_i)
        `EXE_MADD_OP, `EXE_MADDU_OP: begin      // madd、maddu
```

指令

```
if(cnt_i == 2'b00) begin          // 执行阶段第一个时钟周期
```

```
    hilo_temp_o <= mulres;
    cnt_o       <= 2'b01;
    hilo_temp1 <= {`ZeroWord, `ZeroWord};
    stallreq_for_madd_msub <= `Stop;
end else if(cnt_i == 2'b01) begin // 执行阶段第二个时钟周期
```

```
    hilo_temp_o <= {`ZeroWord, `ZeroWord};
    cnt_o       <= 2'b10;
    hilo_temp1 <= hilo_temp_i + {HI, LO};
    stallreq_for_madd_msub <= `NoStop;
end
end
```

```

`EXE_MSUB_OP, `EXE_MSUBU_OP: begin // msub、msubu指令
 令

  if(cnt_i == 2'b00) begin // 执行阶段第一个时钟周期
    hilo_temp_o <= ~mulres + 1 ;
    cnt_o       <= 2'b01;
    stallreq_for_madd_msub <= `Stop;
  end else if(cnt_i == 2'b01)begin // 执行阶段第二个时钟周期
    hilo_temp_o <= {`ZeroWord,`ZeroWord};
    cnt_o       <= 2'b10;
    hilo_temp1  <= hilo_temp_i + {HI,LO};
    stallreq_for_madd_msub <= `NoStop;
  end
  default: begin
    hilo_temp_o <= {`ZeroWord,`ZeroWord};
    cnt_o       <= 2'b00;
    stallreq_for_madd_msub <= `NoStop;
  end
  endcase
end

```

```
end
```

```
/******  
***** 第三段：暂停流水线 *****  
******/
```

```
// 目前只有乘累加、乘累减指令会导致流水线暂停，所以stallreq就直接等于  
// stallreq_for_madd_msub的值  
always @ (*) begin  
    stallreq = stallreq_for_madd_msub;  
end
```

```
.....
```

```
/******  
***** 第四段：修改HI、LO寄存器的写信息 *****  
******/
```

```
always @ (*) begin  
    if(rst == `RstEnable) begin  
        whilo_o <= `WriteDisable;  
        hi_o      <= `ZeroWord;  
        lo_o      <= `ZeroWord;  
    end else if((aluop_i == `EXE_MSUB_OP) ||  
                (aluop_i == `EXE_MSUBU_OP))
```

```

begin
    whilo_o <= `WriteEnable;
    hi_o     <= hilo_temp1[63:32];
    lo_o     <= hilo_temp1[31:0];
end else if((aluop_i == `EXE_MADD_OP) ||
              (aluop_i == `EXE_MADDU_OP))
begin
    whilo_o <= `WriteEnable;
    hi_o     <= hilo_temp1[63:32];
    lo_o     <= hilo_temp1[31:0];
    .....
endmodule

```

上述代码可以分为四段理解。

(1) 第一段：计算从通用寄存器中读出的两个寄存器的乘法结果，保存在mulres中。

(2) 第二段：以乘累加指令为例进行讲解。乘累减指令与此类似。

- 如果cnt_i为2'b00，表示是乘累加指令的第一个执行周期，此时将乘法结果mulres通过接口hilo_temp_o输出到EX/MEM模块，以便在下一个时钟周期使用。同时，设置变量stallreq_for_madd_msub为Stop，表示乘累加指令请求流水线暂停。

- 如果cnt_i为2'b01，表示是乘累加指令的第二个执行周期，此时EX模块的输入hilo_temp_i就是上一个时钟周期得到的乘法结果，所以将hilo_temp_i与HI、LO寄存器的值相加，得到最终的运算结果，保存到变量hilo_temp1中。同时，设置变量stallreq_for_madd_msub为NoStop，表示乘累加指令执行结束，不再请求流水线暂停。最后，设置cnt_o为2'b10，而不是直接设置为2'b00，目的是：如果因其他原因导致流水线保持暂停，那么由于cnt_o为2'b10，所以EX阶段不再计算，从而防止乘累加指令重复运行。

(3) 第三段：给出信号stallreq的值，目前只有乘累加、乘累减指令会导致流水线暂停，所以stallreq就直接等于变量stallreq_for_madd_msub的值。

(4) 第四段：由于乘累加、乘累减指令要将最终结果写入HI、LO寄存器，所以在第四段给出了对HI、LO寄存器的写信息。

7.8.3 修改EX/MEM模块

参考图7-12可知，EX/MEM模块要增加4个接口，具体如表7-3所示。

表7-3 EX/MEM模块增加的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	hilo_i	64	输入	保存的乘法结果
2	cnt_i	2	输入	下一个时钟周期是执行阶段的第几个时钟周期
3	hilo_o	64	输出	保存的乘法结果
4	cnt_o	2	输出	当前处于执行阶段的第几个时钟周期

EX/MEM模块的代码主要修改如下，完整代码位于本书附带光盘中Code\Chapter7_2目录下的ex_mem.v文件。

```
module ex_mem(
    .....
    // 来自控制模块的信息
    input wire[5:0] stall,
    .....
    // 增加的输入接口
    input wire[`DoubleRegBus] hilo_i,
    input wire[1:0] cnt_i,
    .....
    // 增加的输出接口
    output reg[`DoubleRegBus] hilo_o,
    output reg[1:0] cnt_o
);

// 在流水线执行阶段暂停的时候，将输入信号hilo_i通过输出接口hilo_o送出，
// 输入信号cnt_i通过输出接口cnt_o送出。其余时刻，hilo_o为0，cnt_o也为0
always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        .....
```

```

        hilo_o <= {`ZeroWord, `ZeroWord};

        cnt_o  <= 2'b00;

    end else if(stall[3] == `Stop && stall[4] == `NoStop)

begin

    .....

    hilo_o <= hilo_i;

    cnt_o  <= cnt_i;

end else if(stall[3] == `NoStop) begin

    .....

    hilo_o <= {`ZeroWord, `ZeroWord};

    cnt_o  <= 2'b00;

end else begin

    hilo_o <= hilo_i;

    cnt_o  <= cnt_i;

end

end

endmodule

```

7.8.4 修改OpenMIPS模块

因为上面为EX、EX/MEM模块添加了接口，所以需要修改OpenMIPS模块，以将这些接口连接起来，连接关系如图7-12所示，具体代码不在书中列出，读者可以参考本书附带光盘中Code\Chapter7_2目录下的openmips.v文件。

7.9 测试乘累加、乘累减指令实现效果

本节将通过一个测试程序验证为OpenMIPS添加的乘累加、乘累减指令是否实现正确，测试程序如下，源文件是位于本书附带光盘中Code\Chapter7_2\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.global _start
_start:
    ori $1,$0,0xffff
    sll $1,$1,16
    ori $1,$1,0xffffb      # $1 = -5 为寄存器$1赋初值
    ori $2,$0,6            # $2 = 6 为寄存器$2赋初值

    mult $1,$2             # hi = 0xffffffff
                           # lo = 0xffffffe2

    madd $1,$2             # hi = 0xffffffff
                           # lo = 0xfffffc4
```

```

maddu $1,$2          # hi = 0x5
                      # lo = 0xffffffa6

msub $1,$2           # hi = 0x5
                      # lo = 0xfffffc4

msubu $1,$2          # hi = 0xffffffff
                      # lo = 0xfffffe2

```

连续的4条乘累加、乘累减指令，程序的注释给出了预期执行结果。ModelSim仿真如图7-14所示。从图中可知，乘累加、乘累减指令实现正确，同时，可以观察到乘累加、乘累减指令都需要两个时钟周期执行完毕。

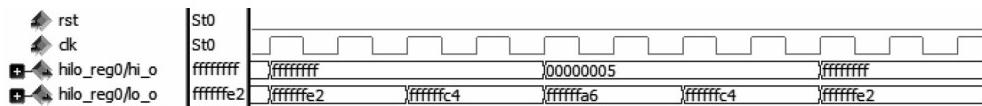


图7-14 乘累加、乘累减指令实现效果

7.10 除法指令说明

除法指令有2条，包括：div、divu，各指令的格式如图7-15所示。从图中可知这2条指令的指令码都是SPECIAL，第6-15bit都为0，可以依据第0-5bit的功能码确定是哪一种指令。

31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000	rs	rt	00000	00000	DIV 011010		div指令
SPECIAL 000000	rs	rt	00000	00000	DIVU 011011		divu指令

图7-15 div、divu指令的格式

- 当功能码是6'b011010时，表示是div指令，有符号除法运算。

指令用法为：div rs, rt。

指令作用为： $\{HI, LO\} \leftarrow rs / rt$ ，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值，作为有符号数进行除法运算，将商保存到寄存器LO，余数保存到寄存器HI。

- 当功能码是6'b011011时，表示是divu指令，无符号除法运算。

指令用法为：divu rs, rt。

指令作用为： $\{HI, LO\} \leftarrow rs / rt$ ，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值，作为无符号数进行除法运算，将商保存到寄存器LO，余数保存到寄存器HI。

7.11 除法指令实现思路

7.11.1 试商法

OpenMIPS设计采用试商法实现除法运算，对于32位的除法，至少需要32个时钟周期才能得到除法结果。本节介绍试商法的一般过程。

设被除数是m，除数是n，商保存在s中，被除数的位数是k，其计算步骤如下（为了便于说明，在此处将所有数据的最低位称为第1位，而不称为第0位）。

1. 取出被除数的最高位 $m[k]$ ，使用被除数的最高位减去除数n，如果结果大于等于0，则商的 $s[k]$ 为1，反之为0。

2. 如果上一步得出的结果是0，表示当前的被减数小于除数，则取出被除数剩下的值的最高位 $m[k-1]$ ，与当前被减数组合作为下一轮的被减数；如果上一步得出的结果是1，表示当前的被减数大于除数，则利用上一步中减法的结果与被除数剩下的值的最高位 $m[k-1]$ 组合作为下一轮的被减数。然后，设置 k 等于 $k-1$ 。

3. 新的被减数减去除数，如果结果大于等于0，则商的 $s[k]$ 为1，否则 $s[k]$ 为0，后面的步骤重复2-3，直到 k 等于1。

上述步骤可以使用图7-16描述。

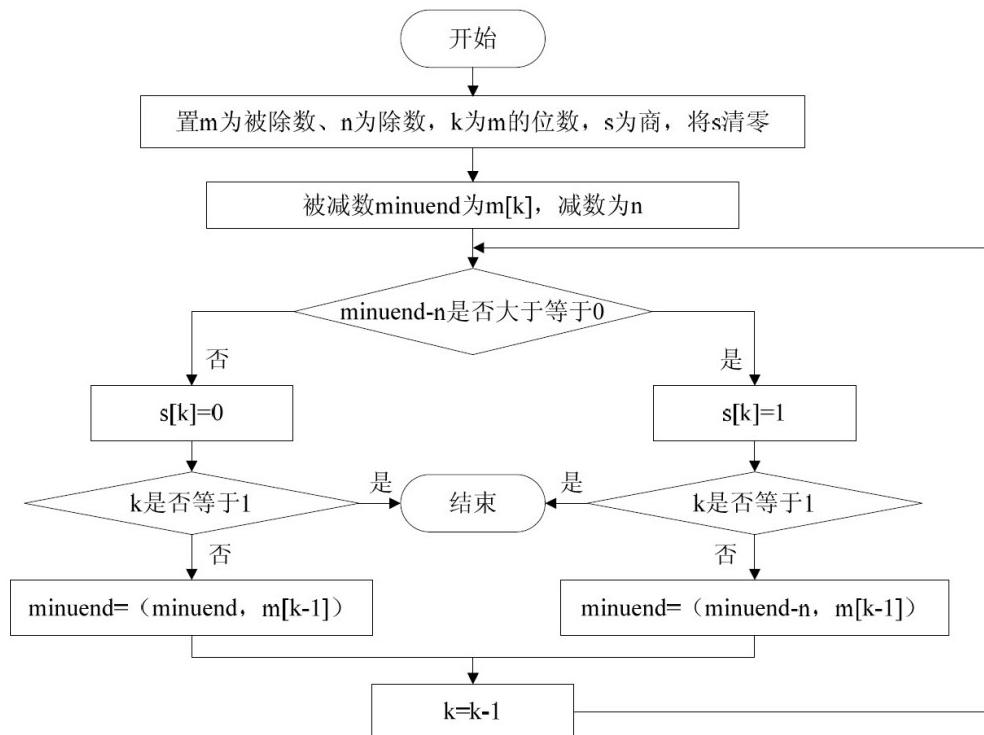


图7-16 试商法的运算过程

以4'b1101除以4'b0010为例，采用试商法时的计算步骤如表7-4所示。

表7-4 使用试商法计算1101/0010（都是二进制）

步 骤		minuend	minuend-n	k	s	说 明
置初值				4	0000	置被除数 m 为 1101，除数 n 为 0010，k 为 4，同时 s 清零
第 0 步	开始时	1	小于 0	4	0000	$(1-0010) < 0, s[4]=0$
	结束时	11		3		新的 minuend= (minuend, m[k-1])
第 1 步	开始时	11	大于 0	3	0100	$(11-0010) > 0, s[3]=1$
	结束时	10		2		新的 minuend= (11-0010, m[k-1]) = 10

续表

步 骤		minuend	minuend-n	k	s	说 明
第 2 步	开始时	10	等于 0	2	0110	$(10-0010) = 0, s[2]=1$
	结束时	01		1		新的 minuend= (10-0010, m[k-1]) = 01
第 3 步	开始时	01	小于 0	1	0110	$(01-0010) < 0, s[1]=0$
结束					0110	最终得到商为 0110

7.11.2 实现思路

新建一个模块DIV，在其中实现采用试商法的32位除法运算。当流水线执行阶段的EX模块发现当前指令是除法指令时，首先暂停流水线，然后将被除数、除数等信息送到DIV模块，开始除法运算。DIV模块在除法运算结束后，通知EX模块，并将除法结果送到EX模块，后者依据除法结果设置HI、LO寄存器的写信息，同时取消暂停流水线。

7.11.3 系统结构的修改

为了实现7.11.2节的思路，修改系统结构如图7-17所示。

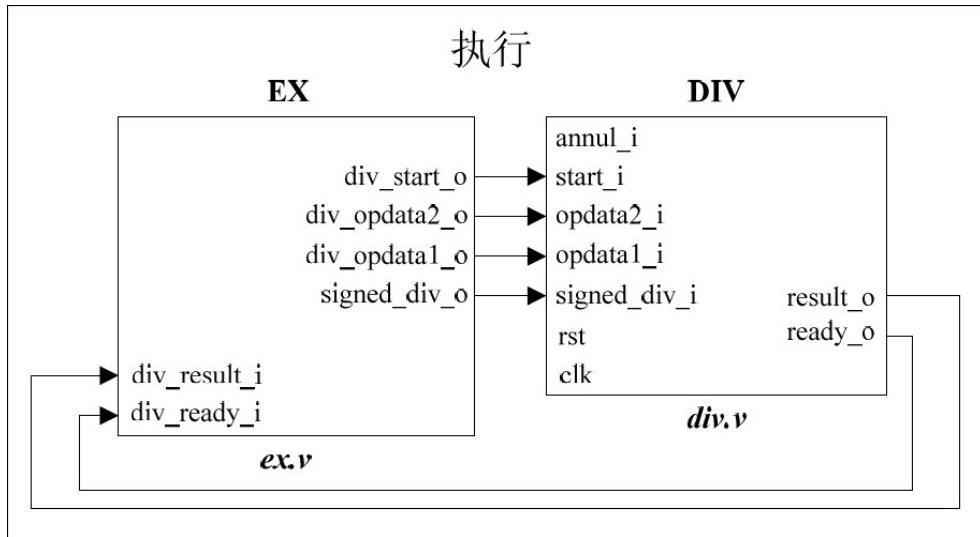


图7-17 为了实现除法指令而对系统结果做的修改

EX模块通过接口div_opdata1_o、div_opdata2_o分别给出被除数、除数，同时通过接口signed_div_o指明是否是有符号除法，然后通过接口div_start_o指示开始除法运算。

DIV模块在除法运行完毕后，通过接口ready_o告知EX模块，并且通过接口result_o输出除法结果，result_o的宽度是64位，其中高32位是余数，低32位是商。

7.12 修改OpenMIPS以实现除法指令

7.12.1 增加DIV模块

DIV模块的接口如图7-17所示，各接口的含义如表7-5所示。

表7-5 DIV模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号, 高电平有效
2	clk	1	输入	时钟信号
3	signed_div_i	1	输入	是否有符号除法, 为 1 表示有符号除法
4	opdata1_i	32	输入	被除数
5	opdata2_i	32	输入	除数
6	start_i	1	输入	是否开始除法运算
7	annul_i	1	输入	是否取消除法运算, 为 1 表示取消除法运算
8	result_o	64	输出	除法运算结果
9	ready_o	1	输出	除法运算是否结束

DIV模块的主要部分是一个状态机，共有四个状态，如下，状态转换如图7-18所示。

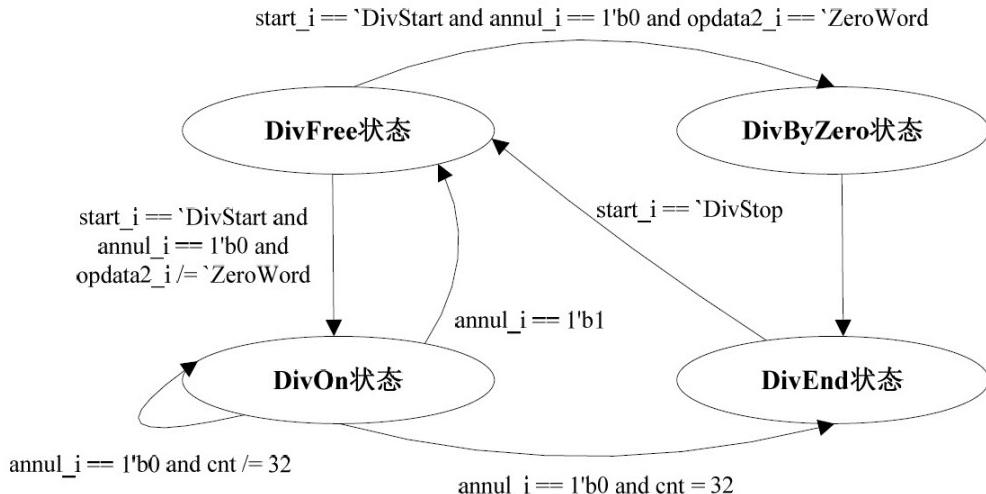


图7-18 DIV模块内部的状态转换图

- DivFree：除法模块空闲。
- DivByZero：除数是0。
- DivOn：除法运算进行中。
- DivEnd：除法运算结束。

复位的时候，DIV模块处于DivFree状态，当输入信号start_i为DivStart，且输入信号annul_i为0时，表示除法操作开始。

- 如果除数opdata2_i为0，那么进入DivByZero状态，直接给出除法结果，这里设置为0，余数也为0，然后进入DivEnd状态，并通知EX模块得到除法运算结果，后者会设置DIV模块的输入信号start_i为DivStop，除法运算结束。
- 如果除数opdata2_i不为0，那么进入DivOn状态，使用试商法，经过32个时钟周期，得出除法结果，然后进入DivEnd状态，并通知EX模块得到除法运算结果，后者会设置DIV模块的输入信号start_i为DivStop，除法运算结束。

DIV模块的代码如下，源文件是本书附带光盘Code\Chapter7_3目录下的div.v。

```
module div(
    input wire clk,
    input wire rst,
    input wire signed_div_i,
    input wire[31:0] opdata1_i,
    input wire[31:0] opdata2_i,
    input wire start_i,
    input wire annul_i,
    output reg[63:0] result_o,
    output reg ready_o
);
    wire[32:0] div_temp;
```

```

reg[5:0]    cnt;           //记录试商法进行了几轮，当等于32时，表示试商法
结束

reg[64:0]   dividend;
reg[1:0]    state;
reg[31:0]   divisor;
reg[31:0]   temp_op1;
reg[31:0]   temp_op2;

//dividend的低32位保存的是被除数、中间结果，第k次迭代结束的时候
dividend[k:0]

//保存的就是当前得到的中间结果，dividend[31:k+1]保存的就是被除数中还没有
参与运算

//的数据，dividend高32位是每次迭代时的被减数，所以dividend[63:32]就是图
7-16

//中的minuend，divisor就是图7-16中的除数n，此处进行的就是minuend-n运
算，结

//果保存在div_temp中

assign div_temp = {1'b0,dividend[63:32]} - {1'b0,divisor};

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        state    <= `DivFree;
        ready_o  <= `DivResultNotReady;
        result_o <= {`ZeroWord,`ZeroWord};
    end else begin
        case (state)
            //*****
            *****          DivFree 状态
            *****

```

```

//分三种情况：

// (1) 开始除法运算，但除数为0，那么进入DivByZero状态

// (2) 开始除法运算，且除数不为0，那么进入DivOn状态，初始化cnt为
0，如

//      果是有符号除法，且被除数或者除数为负，那么对被除数或者除数取
补码。

//      除数保存到divisor中，将被除数的最高位保存到dividend的第
32位，

//      准备进行第一次迭代

// (3) 没有开始除法运算，保持ready_o为DivResultNotReady，保持
//      result_o为0

//*****
`DivFree: begin                                // DivFree状态

if(start_i == `DivStart && annul_i == 1'b0) begin
    if(opdata2_i == `ZeroWord) begin
        state <= `DivByZero;                  // 除数为0
    end else begin
        state <= `DivOn;                     // 除数不为0
    end
end

cnt <= 6'b000000;
if(signed_div_i == 1'b1 && opdata1_i[31] ==
1'b1 ) begin

```

```

        temp_op1 = ~opdata1_i + 1; // 被除数取补码
    end else begin
        temp_op1 = opdata1_i;
    end
    if(signed_div_i == 1'b1 && opdata2_i[31] == 1'b1
) begin
    temp_op2 = ~opdata2_i + 1; // 除数取补码
end else begin
    temp_op2 = opdata2_i;
end
dividend <= {`ZeroWord, `ZeroWord};
dividend[32:1] <= temp_op1;
divisor <= temp_op2;
end
end else begin // 没有开始除法运算

ready_o <= `DivResultNotReady;
result_o <= {`ZeroWord, `ZeroWord};
end
end

//*****          DivByZero 状态
*****
//如果进入DivByZero状态，那么直接进入DivEnd状态，除法结束，且结果
为0
*****
```

```
`DivByZero:      begin          //DivByZero状态

    dividend <= {`ZeroWord, `ZeroWord};
    state <= `DivEnd;
end

//*****
DivOn 状态
*****


//分三种情况:
// (1) 如果输入信号annul_i为1, 表示处理器取消除法运算, 那么DIV模
块直
//      接回到DivFree状态。
// (2) 如果annul_i为0, 且cnt不为32, 那么表示试商法还没有结束, 此
时
//      如果减法结果div_temp为负, 那么此次迭代结果是0, 参考图7-
16; 如
//      果减法结果div_temp为正, 那么此次迭代结果是1, 参考图7-16,
dividend
//      的最低位保存每次的迭代结果。同时保持DivOn状态, cnt加1。
// (3) 如果annul_i为0, 且cnt为32, 那么表示试商法结束, 如果是有符
号
//      除法, 且被除数、除数一正一负, 那么将试商法的结果取补码, 得到
最终的
//      结果, 此处的商、余数都要取补码。商保存在dividend的低32位,
余数
//      保存在dividend的高32位。同时进入DivEnd状态。
```

```
//*****
`DivOn:          begin          //DivOn状态

if(annul_i == 1'b0) begin
    if(cnt != 6'b100000) begin      //cnt不为32，表示试商
法还没有结束
        if(div_temp[32] == 1'b1) begin
//如果div_temp[32]为1，表示 (minuend-n) 结果小于
0,
//将dividend向左移一位，这样就将被除数还没有参与运算
的
//最高位加入到下一次迭代的被减数中，同时将0追加到中间
结果
        dividend <= {dividend[63:0] , 1'b0};

    end else begin
//如果div_temp[32]为0，表示 (minuend-n) 结果大于
等
//于0，将减法的结果与被除数还没有参运算的最高位加入到
下
//一次迭代的被减数中，同时将1追加到中间结果
```

```



```

```

//*****          DivEnd 状态
*****
//除法运算结束, result_o的宽度是64位, 其高32位存储余数, 低32位存
储商,
//设置输出信号ready_o为DivResultReady, 表示除法结束, 然后等待
EX模块
//送来DivStop信号, 当EX模块送来DivStop信号时, DIV模块回到
DivFree
//状态

//*****`DivEnd:      begin      //DivEnd状态
result_o <= {dividend[64:33], dividend[31:0]};
ready_o <= `DivResultReady;
if(start_i == `DivStop
) begin
    state <= `DivFree;
    ready_o <= `DivResultNotReady;
    result_o <= {`ZeroWord, `ZeroWord};
end
end
endcase
end
end

```

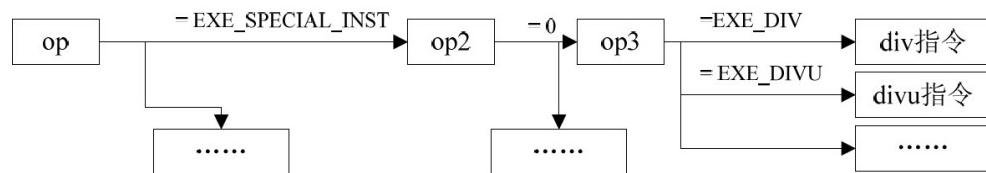
```
endmodule
```

DIV模块中涉及的宏定义，在defines.v中定义，如下：

```
`define DivFree          2'b00
`define DivByZero        2'b01
`define DivOn            2'b10
`define DivEnd           2'b11
`define DivResultReady   1'b1
`define DivResultNotReady 1'b0
`define DivStart          1'b1
`define DivStop           1'b0
```

7.12.2 修改译码阶段的ID模块

译码阶段的ID模块要增加对除法指令的分析，根据图7-15给出的指令格式可知，除法指令都是SPECIAL类指令，可以依据功能码确定是哪一种指令，确定指令的过程如图7-19所示。



```
wire[5:0] op = inst_i[31:26]; wire[4:0] op2 = inst_i[10:6];
wire[5:0] op3 = inst_i[5:0];    wire[4:0] op4 = inst_i[20:16];
```

图7-19 确定除法指令的过程

其中涉及的宏定义如下，正是图7-15中各个指令的功能码。在本书附带光盘Code\Chapter7_3目录下的defines.v文件中可以找到这些定义。

```
`define EXE_DIV    6'b011010  
`define EXE_DIVU   6'b011011
```

修改译码阶段的ID模块如下。完整代码位于本书附带光盘Code\Chapter7_3目录下的id.v文件。

```
module id(  
    .....  
) ;  
    .....  
  
    assign stallreq = `NoStop;  
  
    always @ (*) begin  
        if (rst == `RstEnable) begin  
            .....  
        end else begin  
            aluop_o      <= `EXE_NOP_OP;  
            alusel_o     <= `EXE_RES_NOP;  
            wd_o         <= inst_i[15:11];           // 默认目的寄存器地址  
            wd_o  
            wreg_o       <= `WriteDisable;  
            instinvalid <= `InstInvalid;  
            reg1_read_o <= 1'b0;  
            reg2_read_o <= 1'b0;  
            reg1_addr_o <= inst_i[25:21];           // 默认的reg1_addr_o  
            reg2_addr_o <= inst_i[20:16];           // 默认的reg2_addr_o
```

```

imm          <= `ZeroWord;

case (op)
`EXE_SPECIAL_INST: begin
    case (op2)
        5'b00000: begin
            case (op3)
                .....
`EXE_DIV: begin //div指令

wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_DIV_OP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end

`EXE_DIVU: begin //divu指令

wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_DIVU_OP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end

.....

```

这2条除法指令的译码过程都是相似的，简要说明如下。

(1) 因为最终结果是写入HI、LO寄存器，不需要写通用寄存器，所以赋值wreg_o为WriteDisable。

(2) 因为要读取两个通用寄存器的值，所以设置reg1_read_o、reg2_read_o为1'b1，读取的是图7-15中地址为rs、rt的寄存器的值。

(3) alusel_o的值保持为默认值EXE_RES_NOP。

(4) 设置aluop_o的值与具体指令对应。

7.12.3 修改执行阶段的EX模块

参考图7-17可知，EX模块需要增加部分接口，增加的接口如表7-6所示。

表7-6 EX模块新增的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	signed_div_o	1	输出	是否是有符号除法，为1表示是有符号除法
2	div_opdata1_o	32	输出	被除数
3	div_opdata2_o	32	输出	除数
4	div_start_o	1	输出	是否开始除法运算
5	div_result_i	64	输入	除法运算结果
6	div_ready_i	1	输入	除法运算是否结束

EX模块的代码主要修改如下。完整代码位于本书附带光盘Code\Chapter7_3目录下的ex.v文件。

```
module ex(
    .....
    // 新增来自除法模块的输入
```

```
input wire[`DoubleRegBus]    div_result_i,
input wire                      div_ready_i,
.
.
.

// 新增到除法模块的输出

output reg[`RegBus]           div_opdata1_o,
output reg[`RegBus]           div_opdata2_o,
output reg                   div_start_o,
output reg                   signed_div_o,

output reg                   stallreq

);

.
.

reg stallreq_for_div;          // 是否由于除法运算导致流水线暂停

.
.

*****
***** 第一段：输出DIV模块控制信息，获取DIV模块给出的结果 *****
*****
```

```

    div_opdata1_o      <= `ZeroWord;
    div_opdata2_o      <= `ZeroWord;
    div_start_o        <= `DivStop;
    signed_div_o       <= 1'b0;

end else begin
    stallreq_for_div <= `NoStop;
    div_opdata1_o      <= `ZeroWord;
    div_opdata2_o      <= `ZeroWord;
    div_start_o        <= `DivStop;
    signed_div_o       <= 1'b0;

case (aluop_i)
`EXE_DIV_OP:           begin          //是div指令
    if(div_ready_i == `DivResultNotReady) begin
        div_opdata1_o      <= reg1_i;           //被除数
        div_opdata2_o      <= reg2_i;           //除数
        div_start_o        <= `DivStart;         //开始除法运算
        signed_div_o       <= 1'b1;             //有符号除法
        stallreq_for_div <= `Stop;              //请求流水线暂停
    end else if(div_ready_i == `DivResultReady) begin
        div_opdata1_o      <= reg1_i;
        div_opdata2_o      <= reg2_i;
        div_start_o        <= `DivStop;          //结束除法运算
        signed_div_o       <= 1'b1;
        stallreq_for_div <= `NoStop;            //不再请求流水线暂停
    end
end

```

```

    end else begin
        div_opdata1_o      <= `ZeroWord;
        div_opdata2_o      <= `ZeroWord;
        div_start_o        <= `DivStop;
        signed_div_o       <= 1'b0;
        stallreq_for_div <= `NoStop;
    end
end

`EXE_DIVU_OP:           begin               //是divu指令

```

```

if(div_ready_i == `DivResultNotReady) begin
    div_opdata1_o      <= reg1_i;
    div_opdata2_o      <= reg2_i;
    div_start_o        <= `DivStart;
    signed_div_o       <= 1'b0;           //无符号除法
    stallreq_for_div <= `Stop;
end else if(div_ready_i == `DivResultReady) begin
    div_opdata1_o      <= reg1_i;
    div_opdata2_o      <= reg2_i;
    div_start_o        <= `DivStop;
    signed_div_o       <= 1'b0;
    stallreq_for_div <= `NoStop;
end else begin
    div_opdata1_o      <= `ZeroWord;
    div_opdata2_o      <= `ZeroWord;
    div_start_o        <= `DivStop;
    signed_div_o       <= 1'b0;

```

```

        stallreq_for_div <= `NoStop;

    end
end

default: begin
    end
endcase
end

end

//*****************************************************************************
***** 第二段: 暂停流水线 *****
//*************************************************************************/

```

```

always @ (*) begin
    stallreq = stallreq_for_madd_msub || stallreq_for_div;

```

```

end

.....

```

```

//*****************************************************************************
***** 第三段: 修改HI、LO寄存器写信息 *****
//*************************************************************************/

```

```

always @ (*) begin
    if(rst == `RstEnable) begin
        whilo_o <= `WriteDisable;

```

```

        hi_o      <= `ZeroWord;
        lo_o      <= `ZeroWord;
        .....
        end else if((aluop_i == `EXE_DIV_OP) || (aluop_i ==
`EXE_DIVU_OP)

) begin
    whilo_o <= `WriteEnable;
    hi_o     <= div_result_i[63:32];
    lo_o     <= div_result_i[31:0];
    .....

```

上面的代码可以分为三段理解。

(1) 第一段：如果是div指令，并且DIV模块没有声明除法结束（即div_ready_i等于DivResultNotReady），那么输出被除数、除数、除法开始信号、有符号除法等信息到DIV模块，设置div_start_o为DivStart，以指示DIV模块开始除法运算；同时，设置stallreq_for_div为Stop，表示由于除法运算请求流水线暂停。反之，如果DIV模块声明除法结束（即div_ready_i等于DivResultReady），那么设置div_start_o为DivStop，以指示DIV模块停止除法运算；同时，设置stallreq_for_div为NoStop，表示不是由于除法运算请求流水线暂停。

divu指令的执行过程与div指令类似。

(2) 第二段：给出暂停流水线请求信号stallreq的值，目前已实现的乘累加、乘累减、除法指令都会请求流水线暂停，所以stallreq等于stallreq_for_madd_msub与stallreq_for_div进行逻辑“或”运算的结果。

(3) 第三段：由于除法指令要将最终结果写入HI、LO寄存器，所以在第三段给出了对HI、LO寄存器的写信息。其中div_result_i就是DIV模块计算出来的除法结果，高32位存储的是余数，低32位存储的是商。

7.12.4 修改OpenMIPS模块

因为添加了DIV模块，并且修改了EX模块的接口，所以要修改OpenMIPS顶层模块，以将这些新增模块、接口连接起来，连接关系如图7-17所示。完整代码可以参考本书附带光盘Code\Chapter7_3目录下的openmips.v文件，书中只给出DIV模块的例化语句，如下。

```
div div0(
    .clk(clk),
    .rst(rst),
    .signed_div_i(signed_div),
    .opdata1_i(div_opdata1),
    .opdata2_i(div_opdata2),
    .start_i(div_start),
    .annul_i(1'b0),
    .result_o(div_result),
    .ready_o(div_ready)
);
```

这里需要说明一点，DIV模块的输入接口annul_i在目前固定为0，表示不会有取消除法指令的情况发生，但是在后续章节，当我们实现异常处理的时候，会重新确定DIV模块的输入接口annul_i的值。

7.13 测试除法指令实现效果

本节将通过一个测试程序验证为OpenMIPS添加的除法指令是否实现正确，测试程序如下，源文件是本书附带光盘Code\Chapter7_3\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.global _start

_start:
    ori $2,$0,0xffff
    sll $2,$2,16
    ori $2,$2,0xffff1      # $2 = -15      为寄存器$2赋初值
    ori $3,$0,0x11         # $3 = 17      为寄存器$3赋初值

    div $zero,$2,$3        # hi = 0xffffffff
                           # lo = 0x0

    divu $zero,$2,$3       # hi = 0x00000003
                           # lo = 0x0f0f0f0e

    div $zero,$3,$2        # hi = 2
                           # lo = 0xffffffff
```

给寄存器\$2赋初值-15，寄存器\$3赋初值17，然后分别使用div、divu、div指令进行运算，结果保存在HI、LO寄存器，程序的注释给出了预期执行结果。ModelSim仿真如图7-20所示，从中可知OpenMIPS正确实现了除法指令，并且可以观察到，除法指令需要多个时钟周期才能完成。

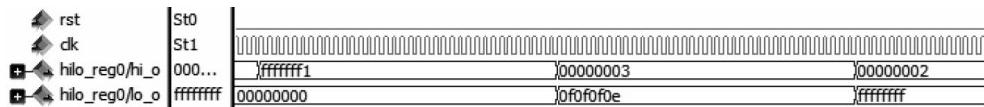


图7-20 ModelSim仿真得到的div、divu指令实现效果

7.14 数据流图的修改

通过本章的工作，我们的OpenMIPS处理器可以执行所有的算术操作指令了，此时的数据流图如图7-21所示。

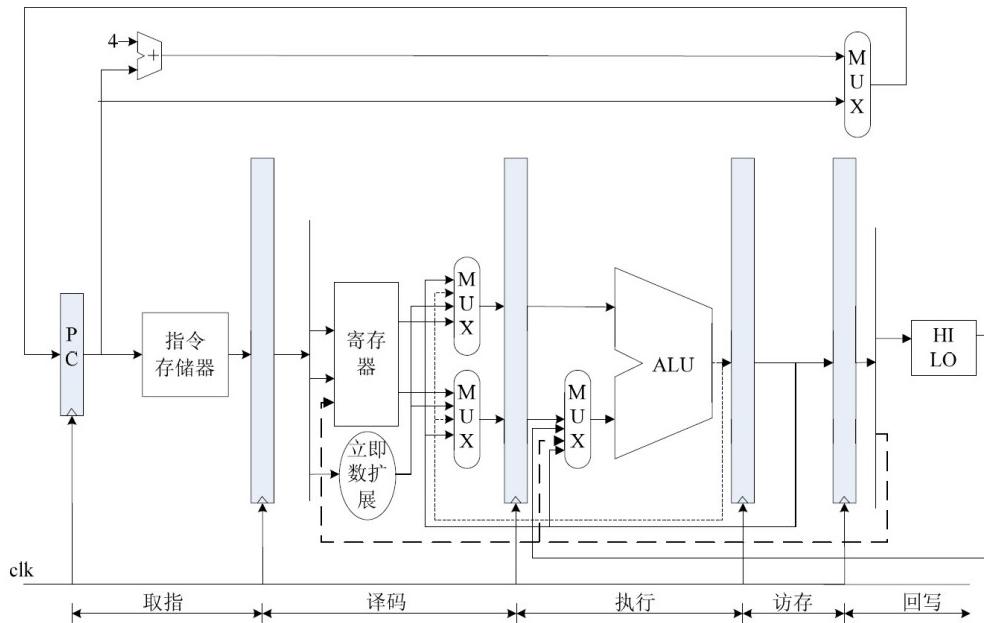


图7-21 增加算术操作后的数据流图

相比第6章的图6-4，主要变化是增加了一个选择器，用来确定PC的值。PC在下一个时钟周期的值可以是 $PC+4$ ，也可以保持当前的值不变，后者对应的就是流水线暂停时的情况。

第8章 转移指令的实现

本章将为OpenMIPS处理器添加转移指令，转移指令包括跳转、分支两种，区别在于前者是绝对转移，后者是相对转移，但实现方法是相似的。转移指令涉及延迟槽，所以首先在8.1节介绍延迟槽的概念，接着在8.2节对MIPS32指令集架构中定义的所有转移指令的格式、作用、用法进行了说明。在8.3节介绍OpenMIPS实现转移指令的思路，以及对数据流图、系统结构的修改。8.4节通过修改代码实现转移指令，最后通过两个测试程序，验证转移指令是否实现正确。

8.1 延迟槽

在实现转移指令之前，先介绍一下延迟槽的概念。在第5章已经介绍了流水线中存在的三种相关：数据相关、结构相关、控制相关。其中控制相关是指流水线中的转移指令或者其他需要改写PC的指令造成的关系。这些指令改写了PC的值，所以导致后面已经进入流水线的几条指令无效，比如：如果转移指令在流水线的执行阶段进行转移条件判断，在发生转移时，会导致当前处于取指、译码阶段的指令无效，需要重新取指。如图8-1所示。

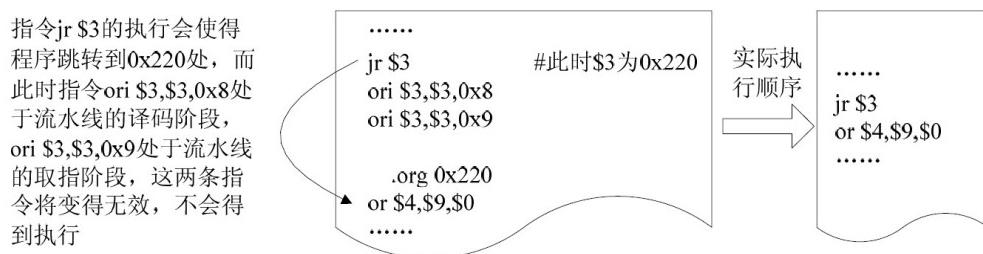


图8-1 转移指令会使得其后面已经进入流水线的几条指令无效

也就是说，在流水线执行阶段进行转移判断，并且转移发生，那么会有2条无效指令，导致浪费了两个时钟周期。为了减少损失，规定转移指令后面的指令位置为“延迟槽”，延迟槽中的指令被称为“延迟指令”（也可称之为“延迟槽指令”）。延迟指令总是被执行，与转移发生与否没有关系。引入延迟槽后的指令执行顺序如图8-2所示。OpenMIPS处理器就计划使用延迟槽技术。

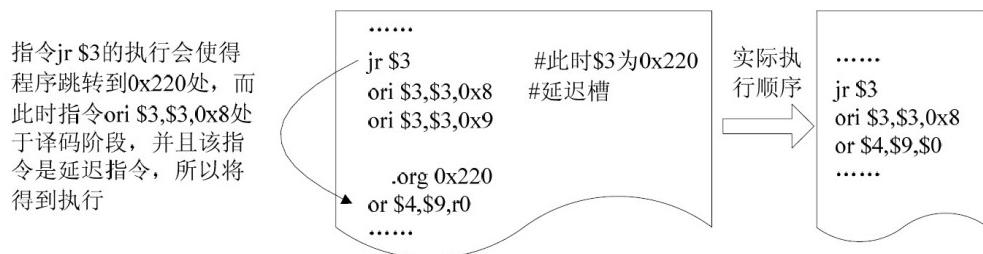


图8-2 引入延迟槽以减少转移带来的损失

但是，即使引入延迟槽，在转移发生时仍然会导致已经进入取指阶段的指令无效，也就是说，仍浪费一个时钟周期，要解决这个问题，可以在译码阶段进行转移判断，这样就可以避免浪费时钟周期。OpenMIPS处理器就设计为在译码阶段进行转移判断。

8.2 转移指令说明

MIPS32指令集架构中定义的转移指令共有14条，可分为如下两类。

- 跳转指令：jr、jalr、j、jal。
- 分支指令：b、bal、beq、bgez、bgezal、bgtz、blez、bltz、bltzal、bne。

其中，跳转指令是绝对转移，分支指令是相对转移。本节分别介绍这两类指令。

1. 跳转指令

跳转指令的格式如图8-3所示。

31 26 25 21 20 16 15 11 10 6 5 0										
SPECIAL 000000	rs	00000	00000	00000	JR 001000	jr指令				
SPECIAL 000000	rs	00000	rd	00000	JALR 001001	jalr指令				
J 000010	instr_index						j指令			
JAL 000011	instr_index						jal指令			

图8-3 跳转指令的格式

从图8-3可知，j、jal指令可以通过指令码进行判断，jr、jalr指令的指令码为SPECIAL，还需要依据功能码进一步判断。

- 当指令中的指令码为SPECIAL，功能码为6'b001000时，表示jr指令。

指令用法为：jr rs。

指令作用为：pc <- rs，将地址为rs的通用寄存器的值赋给寄存器PC，作为新的指令地址。

- 当指令中的指令码为SPECIAL，功能码为6'b001001时，表示jalr指令。

指令用法为：jalr rs 或者jalr rd, rs。

指令作用为： $rd \leftarrow \text{return_address}$, $pc \leftarrow rs$ ， 将地址为rs的通用寄存器的值赋给寄存器PC， 作为新的指令地址， 同时将跳转指令后面第2条指令的地址作为返回地址保存到地址为rd的通用寄存器， 如果没有在指令中指明rd， 那么默认将返回地址保存到寄存器\$31。

- 当指令中的指令码为6'b000010时， 表示j指令。

指令用法为：j target。

指令作用为： $pc \leftarrow (pc+4)[31,28] \parallel \text{target} \parallel '00'$ ， 转移到新的指令地址， 其中新指令地址的低28位是指令中的target（也就是图8-3中的instr_index）左移两位的值， 新指令地址的高4位是跳转指令后面延迟槽指令的地址高4位。

- 当指令中的指令码为6'b000011时， 表示jal指令。

指令用法为：jal target。

指令作用为： $pc \leftarrow (pc+4)[31,28] \parallel \text{target} \parallel '00'$ ， 转移到新的指令地址， 新指令地址与指令j相同， 不再解释。但是， 指令jal还要将跳转指令后面第2条指令的地址作为返回地址保存到寄存器\$31。

j、 jal、 jr、 jalr指令在转移之前都要先执行延迟槽指令。

2. 分支指令

分支指令的格式如图8-4所示。

	31	26 25	21 20	16 15	11 10	6 5	0	
BEQ 000100		rs	rt		offset			beq指令
BEQ 000100		00000	00000		offset			b指令
BGTZ 000111		rs	00000		offset			bgtz指令
BLEZ 000110		rs	00000		offset			blez指令
BNE 000101		rs	rt		offset			bne指令
REGIMM 000001		rs	BLTZ 00000		offset			bltz指令
REGIMM 000001		rs	BLTZAL 10000		offset			bltzal指令
REGIMM 000001		rs	BGEZ 00001		offset			bgez指令
REGIMM 000001		rs	BGEZAL 10001		offset			bgezal指令
REGIMM 000001	00000		BGEZAL 10001		offset			bal指令

图8-4 分支指令的格式

从图8-4可知，前5条指令beq、b、bgtz、blez、bne可以直接依据指令中的指令码进行判断，确定是哪一条指令，而后5条指令bltz、bltzal、bgez、bgezal、bal的指令码都是REGIMM，这是一个宏定义，值为6'b000001，需要根据指令中16-20bit的值进一步判断，从而确定是哪一条指令。

从图8-4还可知，所有分支指令的第0~15bit存储的都是offset，如果发生转移，那么将offset左移2位，并符号扩展至32位，然后与延迟槽指令的地址相加，加法的结果就是转移目的地址，从该地址取指令。

$$\text{转移目标地址} = (\text{signed_extend})(\text{ offset} \parallel '00') + (\text{pc}+4)$$

- 当指令中的指令码为6'b000100时，表示beq指令。

指令用法为：beq rs, rt, offset。

指令作用为：if $rs = rt$ then branch，将地址为rs的通用寄存器的值与地址为rt的通用寄存器的值进行比较，如果相等，那么发生转移。

- 当指令中的指令码为6'b000100，且16-25bit为0时，表示b指令。

指令用法为：b offset。

指令作用为：无条件转移，从图8-4可知，b指令可以认为是beq指令的特殊情况，当beq指令的rs、rt都等于0时，即为b指令，所以在OpenMIPS实现的时候不需要特意实现b指令，只需要实现beq指令即可。

- 当指令中的指令码为6'b000111时，表示bgtz指令。

指令用法为：bgtz rs, offset。

指令作用为：if $rs > 0$ then branch，如果地址为rs的通用寄存器的值大于零，那么发生转移。

- 当指令中的指令码为6'b000110时，表示blez指令。

指令用法为：blez rs, offset。

指令作用为：if $rs \leq 0$ then branch，如果地址为rs的通用寄存器的值小于等于零，那么发生转移。

- 当指令中的指令码为6'b000101时，表示bne指令。

指令用法为： bne rs, rt, offset。

指令作用为： if $rs \neq rt$ then branch，如果地址为rs的通用寄存器的值不等于地址为rt的通用寄存器的值，那么发生转移。

- 当指令中的指令码为REGIMM，且第16~20bit为5'b00000时，表示bltz指令。

指令用法为： bltz rs, offset。

指令作用为： if $rs < 0$ then branch，如果地址为rs的通用寄存器的值小于0，那么发生转移。

- 当指令中的指令码为REGIMM，且第16~20bit为5'b10000时，表示bltzal指令。

指令用法为： bltzal rs, offset。

指令作用为： if $rs < 0$ then branch，如果地址为rs的通用寄存器的值小于0，那么发生转移，并且将转移指令后面第2条指令的地址作为返回地址，保存到通用寄存器\$31。

- 当指令中的指令码为REGIMM，且第16~20bit为5'b00001时，表示bgez指令。

指令用法为： bgez rs, offset。

指令作用为： if $rs \geq 0$ then branch，如果地址为rs的通用寄存器的值大于等于0，那么发生转移。

- 当指令中的指令码为REGIMM，且第16~20bit为5'b10001时，表示bgezal指令。

指令用法为：bgezal rs, offset。

指令作用为：if $rs \geq 0$ then branch，如果地址为rs的通用寄存器的值大于等于0，那么发生转移，并且将转移指令后面第2条指令的地址作为返回地址，保存到通用寄存器\$31。

- 当指令中的指令码为REGIMM，且第21~25bit为0，第16~20bit为5'b10001时，表示bal指令。

指令用法为：bal offset。

指令作用为：无条件转移，并且将转移指令后面第2条指令的地址作为返回地址，保存到通用寄存器\$31。从图8-4的指令格式可知，bal指令是bgezal指令的特殊情况，当bgezal指令的rs为0时，就是bal指令，所以在OpenMIPS实现时，不用特意考虑bal指令，只要实现bgezal指令即可。

综上，b、bal指令不用单独实现，需要OpenMIPS实现的分支指令只有8条。所有的分支指令在转移到目标地址前都要先执行延迟槽中的指令。

8.3 转移指令实现思路

8.3.1 实现思路

根据8.1节的论述，为了尽量减少转移指令带来的损失，OpenMIPS在译码阶段进行转移条件的判断，如果满足转移条件，那么修改PC为转移目标地址。

8.3.2 数据流图的修改

为了实现转移指令，修改数据流图如图8-5所示。

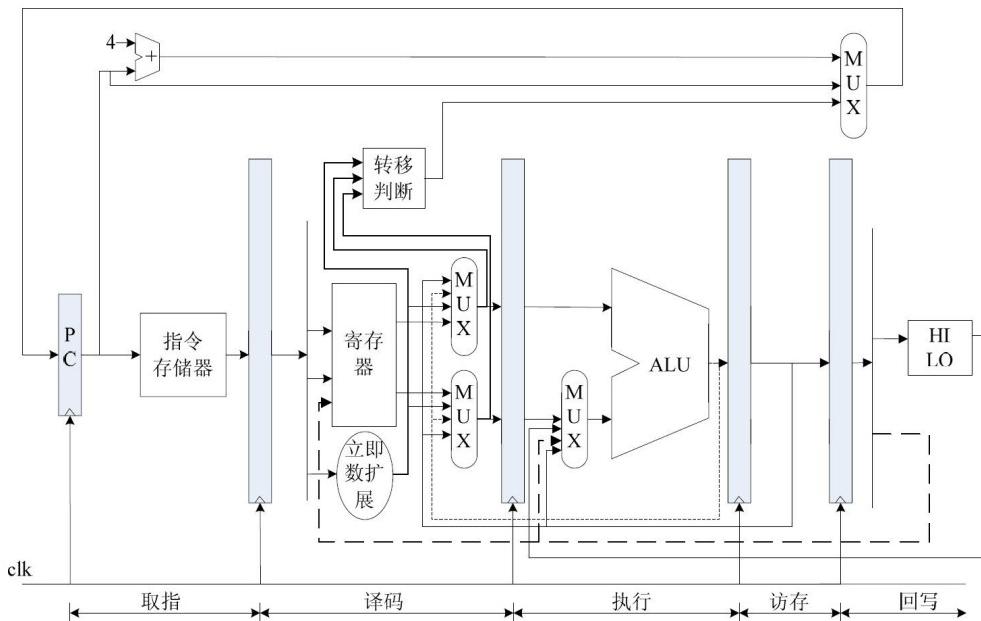


图8-5 为实现转移指令而修改的数据流图

从图8-5中可知，在译码阶段多了转移判断的步骤，此外，PC的取值变为三种情况。

情况一：PC等于 $PC+4$ 。这属于一般情况，每个时钟周期PC加4，指向下一条指令。

情况二：PC保持不变。当流水线暂停的时候，就会发生这种情况，参考第7章中流水线暂停的实现。

情况三：PC等于转移判断的结果。如果是转移指令，且满足转移条件，那么会将转移目标地址赋给PC。

8.3.3 系统结构的修改

为了实现转移指令，需要对系统结构进行修改，增加部分模块的接口，主要修改如图8-6所示。

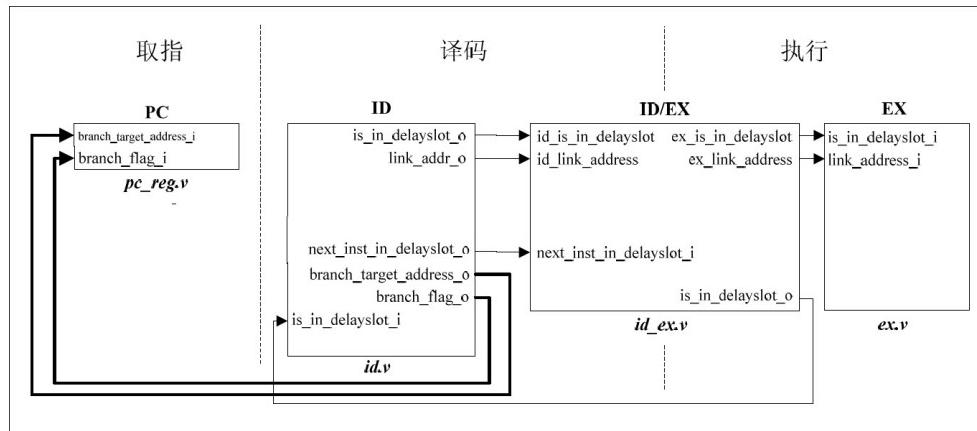


图8-6 为实现转移指令而对系统结构所做的修改

有以下几点说明。

(1) 如果处于译码阶段的指令是转移指令，并且满足转移条件，那么ID模块设置转移发生标志branch_flag_o为Branch，同时通过branch_target_address_o接口给出转移目的地址，送到PC模块，后者据此修改取指地址。

(2) 如果处于译码阶段的指令是转移指令，并且满足转移条件，那么ID模块还会设置next_inst_in_delayslot_o为InDelaySlot，表示下一条指令是延迟槽指令，其中InDelaySlot是一个宏定义。next_inst_in_delayslot_o信号会送入ID/EX模块，并在下一个时钟周期通

过ID/EX模块的is_in_delayslot_o接口送回到ID模块，ID模块可以据此判断当前处于译码阶段的指令是否是延迟槽指令。

(3) 如果转移指令需要保存返回地址，那么ID模块还要计算返回地址，并通过link_addr_o接口输出，该值最终会传递到EX模块，作为要写入目的寄存器的值。

8.4 修改OpenMIPS以实现转移指令

8.4.1 修改取指阶段的PC模块

由图8-6可知，PC模块需要增加接口，增加的接口如表8-1所示。

表8-1 PC模块增加的接口描述

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	branch_flag_i	1	输入	是否发生转移
2	branch_target_address_i	32	输入	转移到的目标地址

修改取指阶段的PC模块如下，主要修改一点：如果branch_flag_i为Branch，那么设置新的PC值为branch_target_address_i。完整代码位于本书附带光盘中Code\Chapter8目录下的pc_reg.v文件中。

```
module pc_reg(  
  
    input wire          clk,  
    input wire          rst,
```

```
// 来自控制模块的信息
input wire[5:0]          stall,
// 来自译码阶段ID模块的信息
           branch_flag_i,

```

```
        end else begin
            pc <= pc + 4'h4;
        end
    end

endmodule
```

其中Branch是defines.v中给出的宏定义：

```
`define Branch      1'b1                  // 转移
`define NotBranch   1'b0                  // 不转移
```

8.4.2 修改译码阶段

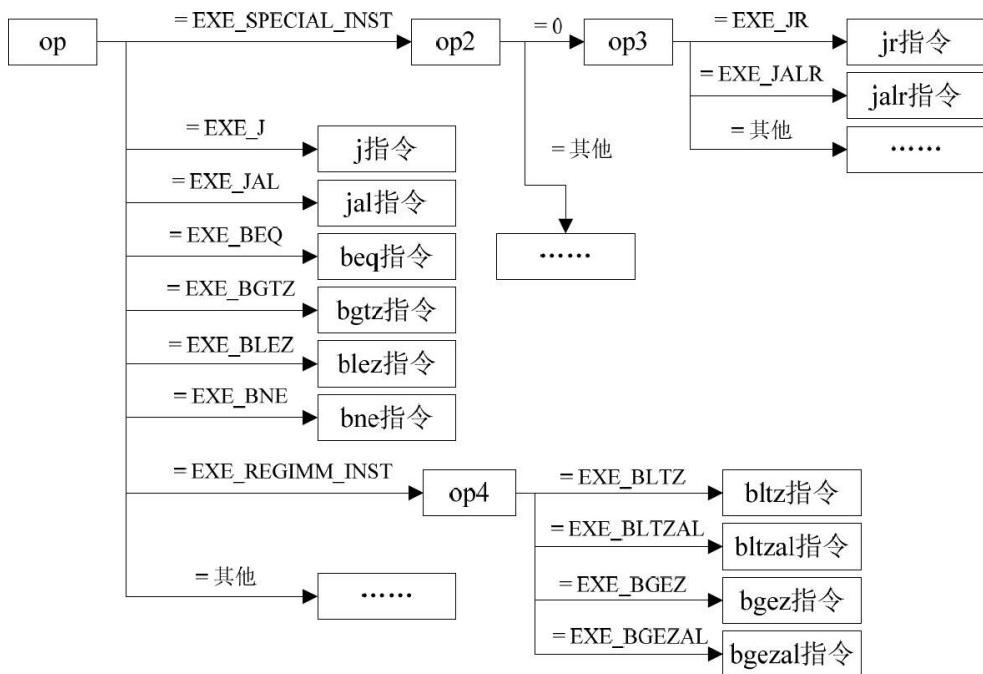
1. 修改ID模块

由图8-6可知，ID模块需要增加一些接口，增加的接口描述如表8-2所示。

表8-2 ID模块新增加的接口描述

序号	接口名	宽度 (bit)	输入/输出	作用
1	branch_flag_o	1	输出	是否发生转移
2	branch_target_address_o	32	输出	转移到的目标地址
3	is_in_delayslot_o	1	输出	当前处于译码阶段的指令是否位于延迟槽
4	link_addr_o	32	输出	转移指令要保存的返回地址
5	next_inst_in_delayslot_o	1	输出	下一条进入译码阶段的指令是否位于延迟槽
6	is_in_delayslot_i	1	输入	当前处于译码阶段的指令是否位于延迟槽

在ID模块要增加对转移指令的分析，根据图8-3、图8-4给出的转移指令格式可得，确定转移指令的过程如图8-7所示。



wire[5:0] op = inst_i[31:26]; wire[4:0] op2 = inst_i[10:6];
 wire[5:0] op3 = inst_i[5:0]; wire[4:0] op4 = inst_i[20:16];

图8-7 确定转移指令的过程

其中涉及的宏定义如下，在本书附带光盘中Code\Chapter8目录下的defines.v文件中可以找到这些定义。

```
`define EXE_J      6'b000010
`define EXE_JAL    6'b000011
`define EXE_JALR   6'b001001
`define EXE_JR     6'b001000
`define EXE_BEQ    6'b000100
`define EXE_BGEZ   5'b00001
`define EXE_BGEZAL 5'b10001
`define EXE_BGTZ   6'b000111
`define EXE_BLEZ   6'b000110
`define EXE_BLTZ   5'b00000
`define EXE_BLTZAL 5'b10000
`define EXE_BNE    6'b000101
```

此外，还新增如下宏定义，在实现转移指令时会使用到：

```
`define InDelaySlot      1'b1          // 在延迟槽中
`define NotInDelaySlot   1'b0          // 不在延迟槽中
```

修改译码阶段的ID模块如下。完整代码请参考本书附带光盘中Code\Chapter8目录下的id.v文件。

```
module id(
  .....
  // 如果上一条指令是转移指令，那么下一条指令进入译码阶段的时候，输入变量
  // is_in_delayslot_i为true，表示是延迟槽指令，反之，为false
  input wire           is_in_delayslot_i,
  .....
```

```

output reg          next_inst_in_delayslot_o,
output reg          branch_flag_o,
output reg[`RegBus] branch_target_address_o,
output reg[`RegBus] link_addr_o,
output reg          is_in_delayslot_o,
.....
);

.....
wire[`RegBus] pc_plus_8;
wire[`RegBus] pc_plus_4;

wire[`RegBus] imm_sll2_signedext;

assign pc_plus_8 = pc_i + 8;      //保存当前译码阶段指令后面第2条指令
的地址
assign pc_plus_4 = pc_i + 4;      //保存当前译码阶段指令后面紧接着的指
令的地址

// imm_sll2_signedext对应分支指令中的offset左移两位，再符号扩展至32位
的值
assign imm_sll2_signedext = {{14{inst_i[15]}}, inst_i[15:0],
2'b00 };

always @ (*) begin

```

```

if (rst == `RstEnable) begin
    .....
    link_addr_o          <= `ZeroWord;
    branch_target_address_o  <= `ZeroWord;
    branch_flag_o         <= `NotBranch;
    next_inst_in_delayslot_o <= `NotInDelaySlot;
end else begin
    .....
    aluop_o      <= `EXE_NOP_OP;
    alusel_o     <= `EXE_RES_NOP;
    wd_o          <= inst_i[15:11];           // 默认目的寄存器地址
wd_o
    wreg_o       <= `WriteDisable;
    instinvalid <= `InstInvalid;
    reg1_read_o <= 1'b0;
    reg2_read_o <= 1'b0;
    reg1_addr_o <= inst_i[25:21];           // 默认的reg1_addr_o
    reg2_addr_o <= inst_i[20:16];           // 默认的reg2_addr_o
    imm          <= `ZeroWord;
    link_addr_o          <= `ZeroWord;
    branch_target_address_o  <= `ZeroWord;
    branch_flag_o         <= `NotBranch;
    next_inst_in_delayslot_o <= `NotInDelaySlot;
case (op)
    `EXE_SPECIAL_INST: begin
        case (op2)
            5'b00000: begin

```

```

case (op3)
    . . .
`EXE_JR: begin // jr指令
    wreg_o          <= `WriteDisable;
    aluop_o         <= `EXE_JR_OP;
    alusel_o        <=
`EXE_RES_JUMP_BRANCH;
    reg1_read_o     <= 1'b1;
    reg2_read_o     <= 1'b0;
    link_addr_o     <= `ZeroWord;
    branch_target_address_o <= reg1_o;
    branch_flag_o    <= `Branch;
    next_inst_in_delayslot_o <= `InDelaySlot;
    instinvalid      <= `InstInvalid;
end
`EXE_JALR: begin // jalr指令
    wreg_o          <= `WriteEnable;
    aluop_o         <= `EXE_JALR_OP;
    alusel_o        <=
`EXE_RES_JUMP_BRANCH;
    reg1_read_o     <= 1'b1;
    reg2_read_o     <= 1'b0;
    wd_o            <= inst_i[15:11];

```

```

        link_addr_o          <= pc_plus_8;
        branch_target_address_o  <= reg1_o;
        branch_flag_o          <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
        instinvalid            <= `InstValid;

    end

    default: begin
        end
    endcase
end

default: begin
end
endcase
end

.....
`EXE_J: begin // j指令

```

```

wreg_o          <= `WriteDisable;
aluop_o          <= `EXE_J_OP;
alusel_o          <= `EXE_RES_JUMP_BRANCH;
reg1_read_o      <= 1'b0;
reg2_read_o      <= 1'b0;
link_addr_o      <= `ZeroWord;
branch_flag_o      <= `Branch;
next_inst_in_delayslot_o <= `InDelaySlot;
instinvalid            <= `InstValid;

```

```

branch_target_address_o  <=
{pc_plus_4[31:28], inst_i[25:0], 2'b00};

end

`EXE_JAL: begin // jal指令

wreg_o                  <= `WriteEnable;
aluop_o                  <= `EXE_JAL_OP;
alusel_o                 <= `EXE_RES_JUMP_BRANCH;
reg1_read_o               <= 1'b0;
reg2_read_o               <= 1'b0;
wd_o                      <= 5'b11111;
link_addr_o                <= pc_plus_8 ;
branch_flag_o              <= `Branch;
next_inst_in_delayslot_o <= `InDelaySlot;
instinvalid                <= `InstValid;
branch_target_address_o  <=
{pc_plus_4[31:28], inst_i[25:0], 2'b00};

end

`EXE_BEQ: begin // beq指令

wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_BEQ_OP;
alusel_o     <= `EXE_RES_JUMP_BRANCH;
reg1_read_o  <= 1'b1;
reg2_read_o  <= 1'b1;

```

```

    instinvalid    <= `InstInvalid;
    if(reg1_o == reg2_o) begin
        branch_target_address_o    <= pc_plus_4 +
imm_sll2_signedext;
        branch_flag_o              <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
end

`EXE_BGTZ: begin                                // bgtz指令

    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_BGTZ_OP;
    alusel_o    <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    instinvalid <= `InstInvalid;
    if((reg1_o[31] == 1'b0) && (reg1_o != `ZeroWord))
begin
        branch_target_address_o    <= pc_plus_4 +
imm_sll2_signedext;
        branch_flag_o              <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
end

`EXE_BLEZ:      begin                            // blez指令

```

```

        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_BLEZ_OP;
        alusel_o    <= `EXE_RES_JUMP_BRANCH;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        instvalid   <= `InstValid;
        if((reg1_o[31] == 1'b1) || (reg1_o == `ZeroWord))
begin
            branch_target_address_o <= pc_plus_4 +
imm_sll2_signedext;
            branch_flag_o           <= `Branch;
            next_inst_in_delayslot_o <= `InDelaySlot;
end
end
`EXE_BNE: begin                                // bne指令

        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_BLEZ_OP;
        alusel_o    <= `EXE_RES_JUMP_BRANCH;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b1;
        instvalid   <= `InstValid;
        if(reg1_o != reg2_o) begin
            branch_target_address_o <= pc_plus_4 +
imm_sll2_signedext;

```

```

        branch_flag_o          <= `Branch;
        next_inst_in_delayslot_o <= `InDelaySlot;
    end
end

`EXE_REGIMM_INST: begin
    case (op4)
        `EXE_BGEZ: begin // bgez指令
            wreg_o      <= `WriteDisable;
            aluop_o      <= `EXE_BGEZ_OP;
            alusel_o     <= `EXE_RES_JUMP_BRANCH;
            reg1_read_o <= 1'b1;
            reg2_read_o <= 1'b0;
            instvalid   <= `InstValid;
            if(reg1_o[31] == 1'b0) begin
                branch_target_address_o <=
                    pc_plus_4 +
imm_sll2_signedext;
                branch_flag_o <= `Branch;
                next_inst_in_delayslot_o <= `InDelaySlot;
            end
        end
        `EXE_BGEZAL: begin // bgezal

```

指令

```

        wreg_o      <= `WriteEnable;
        aluop_o     <= `EXE_BGEZAL_OP;
        alusel_o    <= `EXE_RES_JUMP_BRANCH;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        link_addr_o <= pc_plus_8;
        wd_o         <= 5'b11111;
        instinvalid <= `InstInvalid;
        if(reg1_o[31] == 1'b0) begin
            branch_target_address_o <=
                pc_plus_4 +
imm_sll2_signedext;
            branch_flag_o <= `Branch;
            next_inst_in_delayslot_o <= `InDelaySlot;
        end
    end
`EXE_BLTZ: begin // bltz指令
        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_BGEZAL_OP;
        alusel_o    <= `EXE_RES_JUMP_BRANCH;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        instinvalid <= `InstInvalid;
        if(reg1_o[31] == 1'b1) begin
            branch_target_address_o <=

```

```
imm_sll2_signedext;
    branch_flag_o <= `Branch;
    next_inst_in_delayslot_o <= `InDelaySlot;
end
end

`EXE_BLTZAL: begin // bltzal指令
    wreg_o      <= `WriteEnable;
    aluop_o      <= `EXE_BGEZAL_OP;
    alusel_o     <= `EXE_RES_JUMP_BRANCH;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    link_addr_o <= pc_plus_8;
    wd_o         <= 5'b11111;
    instvalid   <= `InstValid;
    if(reg1_o[31] == 1'b1) begin
        branch_target_address_o <=
pc_plus_4 +
imm_sll2_signedext;
    branch_flag_o <= `Branch;
    next_inst_in_delayslot_o <= `InDelaySlot;
end
end

default: begin
```

```

        end
    endcase

    .....

// 输出变量is_in_delayslot_o表示当前译码阶段指令是否是延迟槽指令
always @ (*) begin
    if(rst == `RstEnable) begin
        is_in_delayslot_o <= `NotInDelaySlot;
    end else begin
        // 直接等于is_in_delayslot_i
        is_in_delayslot_o <= is_in_delayslot_i;
    end
end
end

```

endmodule

对其中几个典型指令的译码过程解释如下。

(1) jr指令

- jr指令不需要保存返回地址，所以设置wreg_o为WriteDisable，设置返回地址link_addr_o为0，aluop_o保持默认值EXE_NOP_OP，alusel_o保持默认值EXE_RES_NOP。
- jr指令要转移到的目标地址是通用寄存器rs的值，所以需要设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存

器，读取的寄存器地址正是指令中的rs，所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值。

- jr指令是绝对转移，所以设置branch_flag_o为Branch。
- 设置转移目标地址branch_target_address_o为reg1_o，也即是读取出来的通用寄存器rs的值。
- 下一条指令是延迟槽指令，所以设置next_inst_in_delayslot_o为InDelaySlot。

j指令与jr类似，只是转移目标地址不再是通用寄存器的值，所以不需要读取通用寄存器，设置reg1_read_o为0，转移目标地址如下。

```
{pc_plus_4[31:28], inst_i[25:0], 2'b00}
```

(2) jalr指令

- jalr指令需要保存返回地址，所以设置wreg_o为WriteEnable，设置返回地址link_addr_o为当前转移指令后面第2条指令的地址，即 pc_plus_8。此外，还要设置alusel_o 为 EXE_RES_JUMP_BRANCH，设置要写的目的寄存器地址wd_o为指令的第11~15bit，正是图8-3中的rd。
- jalr指令要转移到的目标地址是通用寄存器rs的值，所以需要设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器，读取的寄存器地址正是指令中的rs，所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值。
- jalr指令是绝对转移，所以设置branch_flag_o为Branch。
- 设置转移目的地址branch_target_address_o为reg1_o，也即是读取出来的通用寄存器rs的值。

- 下一条指令是延迟槽指令，所以设置next_inst_in_delayslot_o为InDelaySlot。

jal指令与jalr类似，只是jal指令将返回地址写到寄存器\$31中，所以wd_o直接设置为5'b11111，另外，转移目标地址不再是通用寄存器的值，所以不需要读取通用寄存器，设置reg1_read_o为0，转移目标地址如下。

```
{pc_plus_4[31:28], inst_i[25:0], 2'b00}
```

(3) beq指令

- beq指令不需要保存返回地址，所以设置wreg_o为WriteDisable，设置返回地址link_addr_o为0，aluop_o保持默认值EXE_NOP_OP，alusel_o保持默认值EXE_RES_NOP。
- beq指令是条件转移，转移的条件是两个通用寄存器的值相等，所以需要读取两个通用寄存器，设置reg1_read_o、reg2_read_o为1，表示通过Regfile模块的读端口1、读端口2读取寄存器，读取的寄存器地址分别为指令中的rs、rt。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。
- 对于beq指令，如果读取的两个通用寄存器的值相等（即reg1_o等于reg2_o），那么转移发生，设置branch_flag_o为Branch，同时设置转移目的地址branch_target_address_o为pc_plus_4 + imm_sll2_signedext。此外，下一条指令是延迟槽指令，所以设置next_inst_in_delayslot_o为InDelaySlot。

bne指令与beq类似，只是转移条件是两个通用寄存器的值不相等。

(4) bgtz指令

- bgtz 指令不需要保存返回地址，所以设置wreg_o为WriteDisable，设置返回地址link_addr_o为0，aluop_o保持默认值EXE_NOP_OP，alusel_o保持默认值EXE_RES_NOP。
- bgtz指令是条件转移，转移的条件是地址为rs的通用寄存器的值大于0，所以需要设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器，读取的寄存器地址正是指令中的rs。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值。
- 对于bgtz指令，如果读取的地址为rs的通用寄存器的值大于0（即reg1_o大于0），那么转移发生，设置branch_flag_o为Branch，同时设置转移目的地址branch_target_address_o为pc_plus_4 + imm_sll2_signedext。此外，下一条指令是延迟槽指令，所以设置next_inst_in_delayslot_o为InDelaySlot。

blez、bgez、bltz指令与bgtz类似，只是转移的条件不同。

(5) bgezal指令

- bgezal 指令需要保存返回地址，所以设置wreg_o为WriteEnable，设置返回地址link_addr_o为pc_plus_8，设置alusel_o为EXE_RES_JUMP_BRANCH，此外，要将返回地址保存到寄存器\$31，所以设置wd_o为5'b11111。
- bgezal指令是条件转移，转移的条件是地址为rs的通用寄存器的值大于等于0，所以需要设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器，读取的寄存器地址正是指令中的rs。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值。

- 对于bgezal指令，如果读取的地址为rs的通用寄存器的值大于等于0（即reg1_o大于等于0），那么转移发生，设置branch_flag_o为Branch，同时设置转移目的地址branch_target_address_o为pc_plus_4 + imm_sll2_signedext。此外，下一条指令是延迟槽指令，所以设置next_inst_in_delayslot_o为InDelaySlot。

bltzal指令与bgezal类似，只是转移条件是地址为rs的通用寄存器的值小于0。

2. 修改ID/EX模块

参考图8-6可知，ID/EX模块需要增加一些接口，增加的接口描述如表8-3所示。

表8-3 ID/EX模块新增加的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	id_is_in_delayslot	1	输入	当前处于译码阶段的指令是否位于延迟槽
2	id_link_address	32	输入	处于译码阶段的转移指令要保存的返回地址
3	next_inst_in_delayslot_i	1	输入	下一条进入译码阶段的指令是否位于延迟槽
4	ex_is_in_delayslot	1	输出	当前处于执行阶段的指令是否位于延迟槽
5	ex_link_address	32	输出	处于执行阶段的转移指令要保存的返回地址
6	is_in_delayslot_o	1	输出	当前处于译码阶段的指令是否位于延迟槽

ID/EX模块的代码主要修改如下，很简单，当流水线译码阶段没有被暂停时，ID/EX模块在时钟上升沿将新增加的输入传递到对应的输出。完整代码位于本书附带光盘Code\Chapter8目录下的id_ex.v文件中。

```

module id_ex(
    .....
    input wire[`RegBus]           id_link_address,
    input wire                   id_is_in_delayslot,
    input wire                   next_inst_in_delayslot_i,
    .....
    output reg[`RegBus]          ex_link_address,
    output reg                   ex_is_in_delayslot,
    output reg                   is_in_delayslot_o
);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        .....
        ex_link_address     <= `ZeroWord;
        ex_is_in_delayslot <= `NotInDelaySlot;
        is_in_delayslot_o  <= `NotInDelaySlot;
    end else if(stall[2] == `Stop && stall[3] == `NoStop)
begin
    .....
    ex_link_address     <= `ZeroWord;
    ex_is_in_delayslot <= `NotInDelaySlot;
end else if(stall[2] == `NoStop) begin

```

```

    .....
    ex_link_address      <= id_link_address;

    ex_is_in_delayslot <= id_is_in_delayslot;

    is_in_delayslot_o  <= next_inst_in_delayslot_i;

end
end
.....

```

8.4.3 修改执行阶段的EX模块

由图8-6可知，EX模块需要增加一些接口，增加的接口描述如表8-4所示。

表8-4 EX模块新增加的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	is_in_delayslot_i	1	输入	当前处于执行阶段的指令是否位于延时槽
2	link_address_i	32	输入	处于执行阶段的转移指令要保存的返回地址

EX模块的代码主要修改如下，完整代码请参考本书附带光盘中Code\Chapter8目录下的ex.v文件。

```
module ex(  
    .....  
    // 处于执行阶段的转移指令要保存的返回地址  
    input wire[`RegBus]           link_address_i,  
  
    // 当前执行阶段的指令是否位于延迟槽  
    input wire                   is_in_delayslot_i,  
    .....  
) ;  
    .....  
    always @ (*) begin  
        .....  
        case ( alusel_i )  
            `EXE_RES_LOGIC: begin  
                wdata_o <= logicout;  
            end
```

```

`EXE_RES_SHIFT: begin
    wdata_o <= shiftres;
end

`EXE_RES_MOVE: begin
    wdata_o <= moveres;
end

`EXE_RES_ARITHMETIC: begin
    wdata_o <= arithmeticres;
end

`EXE_RES_MUL: begin
    wdata_o <= mulres[31:0];
end

`EXE_RES_JUMP_BRANCH: begin
    wdata_o <= link_address_i;
end

default: begin
    wdata_o <= `Zeroword;
end

endcase
end

```

```
.....  
endmodule
```

如果alusel_o为EXE_RES_JUMP_BRANCH，那么就将返回地址link_address_i作为要写入目的寄存器的值赋给wdata_o。

注意一点，此处并没有利用输入的信号is_in_delayslot_i，该信号表示当前处于执行阶段的指令是否是延迟槽指令，这个信号会在异常处理过程中使用到，本章暂时不需要。

8.4.4 修改OpenMIPS模块

因为有一些模块添加了接口，所以需要修改顶层模块OpenMIPS，以将这些新增加的接口按照图8-6所示的关系连接起来。具体修改也很简单，不在书中列出，读者可以参考本书附带光盘Code\Chapter8目录下的openmips.v文件。

8.5 测试转移指令的实现效果

本节将通过两个测试程序验证转移指令是否实现正确，这两个测试程序分别验证跳转指令、分支指令。

8.5.1 测试跳转指令

测试代码如下，源文件是本书光盘中Code\Chapter8\AsmTest\Test1目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder      # 添加这个伪操作，指示编译器不要对程序做出任何优化或是改动
.set nomacro
.global _start

_start:
    ori $1,$0,0x0001    # (1) $1 = 0x1
    j    0x20            # 转移到0x20处
    ori $1,$0,0x0002    # (2) $1 = 0x2，这是延迟槽指令
    ori $1,$0,0x1111
    ori $1,$0,0x1100

.org 0x20
    ori $1,$0,0x0003    # (3) $1 = 0x3
    jal   0x40            # 转移到0x40处，同时设置$31为0x2c
    div   $zero,$31,$1     # (4) 此时$31 = 0x2c, $1 = 0x3, 所以得到除法结果
                                # HI = 0x2, LO = 0xe, 这是延迟槽指令

    ori $1,$0,0x0005    # (6) $1 = 0x5
    ori $1,$0,0x0006    # (7) $1 = 0x6
    j    0x60            # 转移到0x60处
    nop
```

```
.org 0x40
jalr $2,$31          # 此时$31为0x2c，所以转移到0x2c，同时设置$2
为0x48
or    $1,$2,$0          # (5) $1 = 0x48, 这是延迟槽指令

ori   $1,$0,0x0009      # (10) $1 = 0x9
ori   $1,$0,0x000a      # (11) $1 = 0xa
j    0x80                # 转移到0x80处
nop

.org 0x60
ori   $1,$0,0x0007      # (8) $1 = 0x7
jr    $2                  # 此时$2为0x48，所以转移到0x48处
ori   $1,$0,0x0008      # (9) $1 = 0x8, 这是延迟槽指令
ori   $1,$0,0x1111
ori   $1,$0,0x1100

.org 0x80
nop

loop:
j _loop
nop
```

上述程序验证了j、jal、jr、jalr指令，程序的注释给出了寄存器\$1的变化情况，注意\$1的变化是按照注释中的序号顺序进行的。

ModelSim仿真结果如图8-8所示，观察\$1的变化可知OpenMIPS正确实现了跳转指令。

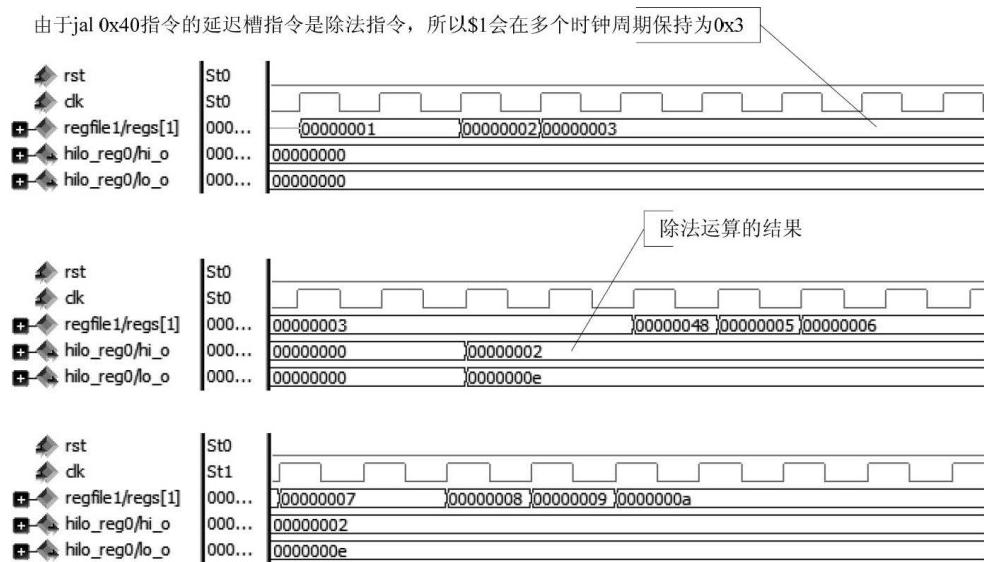


图8-8 跳转指令的仿真测试结果

8.5.2 测试分支指令

测试代码如下，源文件是本书光盘中Code\Chapter8\AsmTest\Test2目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:
    ori  $3,$0,0x8000
    sll  $3,16          # 设置$3 = 0x80000000
```

```

ori $1,$0,0x0001      # (1) $1 = 0x1
b    s1                  # 转移到s1处
ori $1,$0,0x0002      # (2) $1 = 0x2, 这是延迟槽指令

1:
ori $1,$0,0x1111
ori $1,$0,0x1100

.org 0x20

s1:
ori $1,$0,0x0003      # (3) $1 = 0x3
bal s2                  # 转移到s2处, 同时设置$31为0x2c
div $zero,$31,$1       # (4) 此时$31 = 0x2c, $1 = 0x3, 所以除法

```

结果为

```

#      HI = 0x2, LO = 0xe, 这是延迟槽指令

ori $1,$0,0x1100
ori $1,$0,0x1111
bne $1,$0,s3
nop
ori $1,$0,0x1100
ori $1,$0,0x1111

.org 0x50

s2:
ori $1,$0,0x0004      # (5) $1 = 0x4
beq $3,$3,s3           # $3等于$3, 所以会发生转移, 目的地是s3
or   $1,$31,$0          # (6) $1 = 0x2c, 这是延迟槽指令
ori $1,$0,0x1111

```

```
ori $1,$0,0x1100

2:
ori $1,$0,0x0007      # (9) $1 = 0x7
ori $1,$0,0x0008      # (10) $1 = 0x8
bgtz $1,s4            # 此时$1为0x8, 大于0, 所以转移至标号s4处
ori $1,$0,0x0009      # (11) $1 = 0x9, 这是延迟槽指令
ori $1,$0,0x1111
ori $1,$0,0x1100

.org 0x80

s3:
ori $1,$0,0x0005      # (7) $1 = 0x5
bgez $1,2b             # 此时$1为0x5, 大于0, 所以转移至前面的标号2
处
ori $1,$0,0x0006      # (8) $1 = 0x6, 这是延迟槽指令
ori $1,$0,0x1111
ori $1,$0,0x1100

.org 0x100

s4:
ori $1,$0,0x000a      # (12) $1 = 0xa
bgezal $3,s3          # 此时$3为0x80000000, 小于0, 所以不发生转
移
or $1,$0,$31            # (13) $1 = 0x10c
ori $1,$0,0x000b        # (14) $1 = 0xb
ori $1,$0,0x000c        # (15) $1 = 0xc
ori $1,$0,0x000d        # (16) $1 = 0xd
```

```
ori $1,$0,0x000e      # (17) $1 = 0xe
bltz $3,s5           # 此时$3为0x80000000, 小于0, 所以发生转
移, 转移至s5处
ori $1,$0,0x000f      # (18) $1 = 0xf, 这是延迟槽指令
ori $1,$0,0x1100

.org 0x130

s5:
ori $1,$0,0x0010      # (19) $1 = 0x10
blez $1,2b             # 此时$1为0x10, 大于0, 所以不发生转移
ori $1,$0,0x0011      # (20) $1 = 0x11
ori $1,$0,0x0012      # (21) $1 = 0x12
ori $1,$0,0x0013      # (22) $1 = 0x13
bltzal $3,s6          # 此时$3为0x80000000, 小于0, 所以发生转
移, 转移到s6处
or $1,$0,$31           # (23) $1 = 0x14c, 这是延迟槽指令
ori $1,$0,0x1100

.org 0x160

s6:
ori $1,$0,0x0014      # (24) $1 = 0x14
nop
```

```

loop:
    j _loop
    nop

```

上面的测试程序使用了所有的分支指令，程序的注释给出了寄存器\$1的变化情况以及指令执行顺序，注意寄存器\$1的变化是按照注释中的序号顺序进行的。ModelSim仿真结果如图8-9所示，观察\$1的变化可知OpenMIPS正确实现了分支指令。

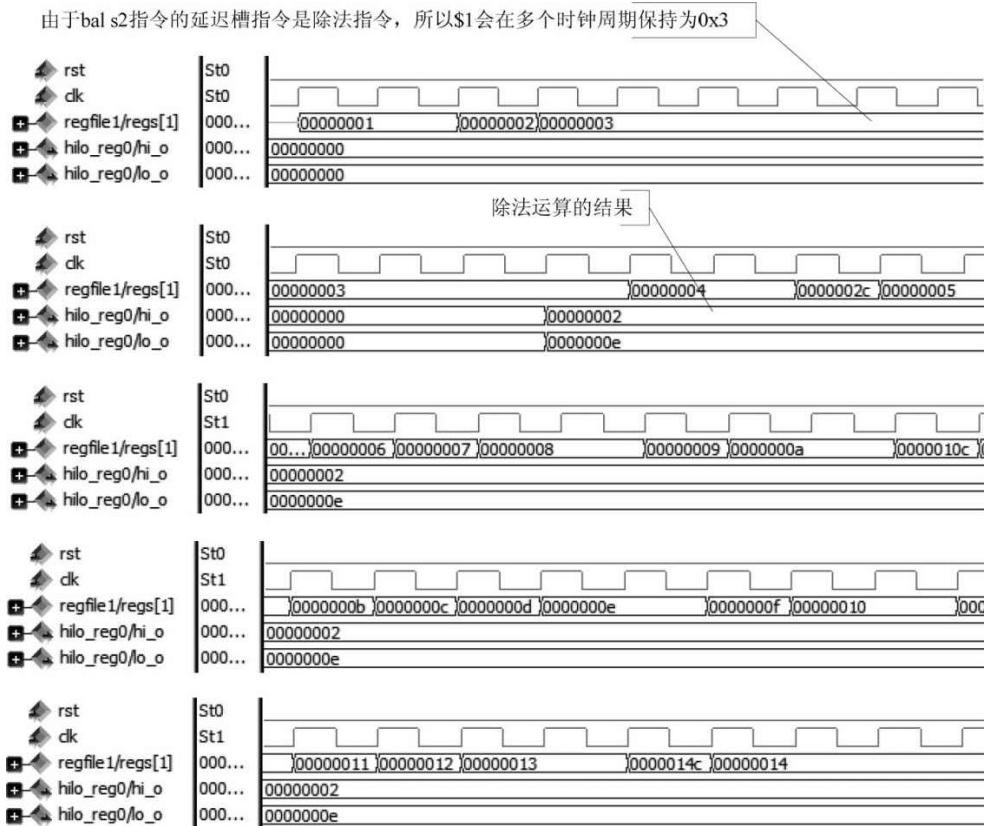


图8-9 分支指令的仿真测试结果

第9章 加载存储指令的实现

本章将实现MIPS32指令集架构中定义的加载存储指令，分两步：首先实现除ll、sc指令外的一般加载存储指令，其次实现比较特殊的加载存储指令ll、sc。

读者可以将本章内容分为五个部分理解阅读：（1）9.1至9.3节介绍了一般加载存储指令的实现；（2）为了验证加载存储指令是否实现正确，在9.4节修改了我们之前一直用来自做测试的SOPC，为其添加了数据RAM；（3）9.5节给出了针对一般加载存储指令的测试程序，通过ModelSim仿真验证指令是否实现正确；（4）9.6至9.9节介绍了特殊加载存储指令ll、sc的实现；（5）9.10至9.12节探讨了由于加载指令引起的load相关问题，给出了OpenMIPS的解决方法，最后验证了解决效果。

9.1 加载存储指令说明

MIPS32指令集架构中定义的加载存储指令共有14条，如下。

- 8条加载指令：lb、lbu、lh、lhu、ll、lw、lwl、lwr。
- 6条存储指令：sb、sc、sh、sw、swl、swr。

对ll、sc指令的说明将放在9.6节，本节介绍其余的12条指令，在本书中也称为一般加载存储指令。其中，由于lwl、lwr、swl、swr这4条指令的作用不太容易理解，所以在9.1.4、9.1.5节专题介绍。

9.1.1 加载指令lb、lbu、lh、lhu、lw说明

加载指令lb、lbu、lh、lhu、lw的格式如图9-1所示。

31	26 25	21 20	16 15	0	
					lb指令
LB 100000	base	rt	offset		lbu指令
LBU 100100	base	rt	offset		lh指令
LH 100001	base	rt	offset		lhu指令
LHU 100101	base	rt	offset		lw指令
LW 100011	base	rt	offset		

图9-1 加载指令lb、lbu、lh、lhu、lw的格式

从图9-1可知，这5条加载指令可以根据指令中26-31bit的指令码加以区分，另外，加载指令的第0~15bit是offset、第21~15bit是base，加载地址的计算方法如下，先将16位的offset符号扩展至32位，然后与地址为base的通用寄存器的值相加，即可得到加载地址。

$$\text{加载地址} = \text{signed_extended(offset)} + \text{GPR}[base]$$

下面分别介绍各个加载指令的作用。

- 当指令中的指令码为6'b100000时，是lb指令，字节加载指令。

指令用法为：lb rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字节，然后符号扩展至32位，保存到地址为rt的通用寄存器中。

- 当指令中的指令码为6'b100100时，是lbu指令，无符号字节加载指令。

指令用法为：lbu rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字节，然后无符号扩展至32位，保存到地址为rt的通用寄存器中。

- 当指令中的指令码为6'b100001时，是lh指令，半字加载指令。

指令用法为：lh rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个半字，然后符号扩展至32位，保存到地址为rt的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低位为0。

- 当指令中的指令码为6'b100101时，是lhu指令，无符号半字加载指令。

指令用法为：lhu rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个半字，然后无符号扩展至32位，保存到地址为rt的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低位为0。

- 当指令中的指令码为6'b100011时，是lw指令，字加载指令。

指令用法为：lw rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字，保存到地址为rt的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低

两位为00。

9.1.2 存储指令sb、sh、sw说明

存储指令sb、sh、sw的格式如图9-2所示。

31	26 25	21 20	16 15	0	
	SB 101000	base	rt	offset	sb指令
	SH 101001	base	rt	offset	sh指令
	SW 101011	base	rt	offset	sw指令

图9-2 存储指令的格式

从图9-2可知，这3条存储指令可以根据指令中26~31bit的指令码加以区分，另外，存储指令的第0~15bit是offset、第21~15bit是base，存储地址的计算方法如下，先将16位的offset符号扩展至32位，然后与地址为base的通用寄存器的值相加，即可得到存储地址。

$$\text{存储地址} = \text{signed_extended(offset)} + \text{GPR}[base]$$

下面分别介绍各个存储指令的作用。

- 当指令中的指令码为6'b101000时，是sb指令，字节存储指令。

指令用法为：sb rt, offset(base)。

指令作用为：将地址为rt的通用寄存器的最低字节存储到内存中的指定地址。

- 当指令中的指令码为6'b101001时，是sh指令，半字存储指令。

指令用法为：sh rt, offset(base)。

指令作用为：将地址为rt的通用寄存器的最低两个字节存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低位为0。

- 当指令中的指令码为6'b101011时，是sw指令，字存储指令。

指令用法为：sw rt, offset(base)。

指令作用为：将地址为rt的通用寄存器的值存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低两位为00。

9.1.3 加载存储指令用法示例

OpenMIPS处理器是按照字节寻址，并且是大端模式，在这种模式下，数据的高位保存在存储器的低地址中，而数据的低位保存在存储器的高地址中。比如：使用指令sb在0x50处存储0x81，存储器中实际存储效果如图9-3所示。

地址	0x50	0x51	0x52	0x53
数据	0x81	0	0	0

图9-3 使用指令sb在0x50处存储0x81

使用指令sh在0x54处存储0x8281，存储器中实际存储效果如图9-4所示。

地址	0x54	0x55	0x56	0x57
数据	0x82	0x81	0	0

图9-4 使用指令sh在0x54处存储0x8281

使用指令sw在0x58处存储0x84838281，存储器中实际存储效果如图9-5所示。

地址	0x58	0x59	0x5a	0x5b
数据	0x84	0x83	0x82	0x81

图9-5 使用指令sw在0x58处存储0x84838281

此时使用加载指令会有如下效果。

(1) 使用指令lbu从0x58处加载一个字节，读出的字节就是0x84，经无符号扩展至32位是0x00000084。

(2) 使用指令lb从0x58处加载一个字节，读出的字节就是0x84，经符号扩展至32位是0xfffffff84。

(3) 使用指令lhu从0x58处加载一个半字，读出的半字就是0x8483，经无符号扩展至32位是0x00008483。

(4) 使用指令lh从0x58处加载一个半字，读出的半字就是0x8483，经符号扩展至32位是0xffff8483。

(5) 使用指令lh从0x59处加载一个半字，不满足地址对齐要求，会出现异常。

(6) 使用指令lhu从0x5a处加载一个半字，读出的半字就是0x8281，经无符号扩展至32位是0x00008281。

(7) 使用指令lh从0x5a处加载一个半字，读出的半字就是0x8281，经符号扩展至32位是0xffff8281。

(8) 使用指令lw从0x58处加载一个字，读出的字就是0x84838281。

9.1.4 加载指令lwl、lwr说明

加载指令lwl、lwr的格式如图9-6所示。

31	26 25	21 20	16 15	0	
LWL 100010	base	rt	offset		lwl指令
LWR 100110	base	rt	offset		lwr指令

图9-6 加载指令lwl、lwr的格式

- 当指令中的指令码为6'b100010时，是lwl指令，非对齐加载指令，向左加载。

指令用法为：lwl rt, offset(base)。

指令作用为：从内存中指定的加载地址处，加载一个字的最高有效部分。lwl指令对加载地址没有要求，从而允许地址非对齐加载，这是与前面介绍的lh、lhu、lw指令的不同之处。在大端模式、小端模式

下，lw1指令的效果不同，因为OpenMIPS是大端模式，所以此处只介绍在大端模式下lw1指令的效果。假设计算出来的加载地址是loadaddr，loadaddr的最低两位的值为n，将loadaddr最低两位设为0后的值称为loadaddr_align，如下。

加载地址 $\text{loadaddr} = \text{signed_extended}(\text{offset}) + \text{GPR}[\text{base}]$

n = loadaddr[1:0]

loadaddr_align = loadaddr - n

例如：假设计算出来的加载地址是5，lw1指令要从地址5加载数据，那么loadaddr就等于5，n等于1，loadaddr_align等于4。

lw1指令的作用是从地址为loadaddr_align处加载一个字，也就是4个字节，然后将这个字的最低4-n个字节保存到地址为rt的通用寄存器的高位，并且保持低位不变。

继续上例，此时loadaddr_align为4，所以从地址4处加载一个字，对应的是地址为4、5、6、7的字节，因为n等于1，所以将加载到的字的最低3个字节保存到地址rt的通用寄存器的高3个字节。如图9-7所示。一个更加通用的描述如图9-8所示。

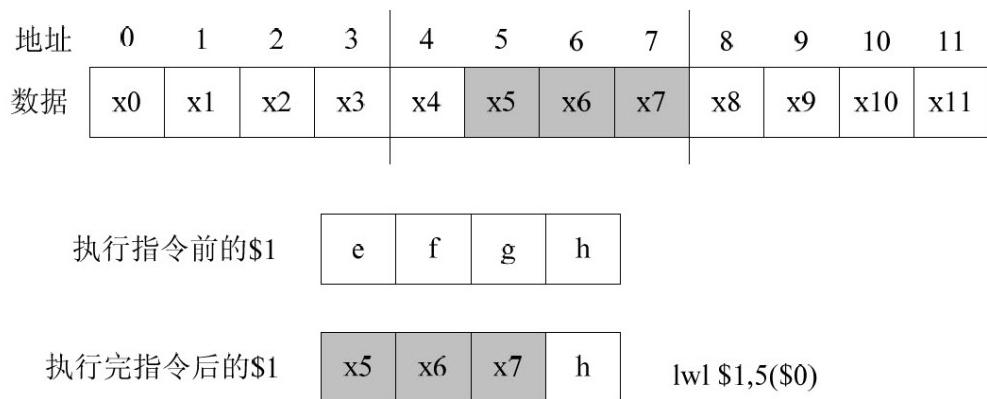


图9-7 lw1指令作用举例

地址loadaddr_align对应的字				执行完lw1指令后， 目标寄存器的值			
I J K L				I J K L			
e f g h				J K L h			
K L g h				L f g h			

继续上例，此时loadaddr_align为8，所以从地址8处加载一个字，对应的是地址为8、9、10、11的字节，因为n等于1，所以将加载到的字的最高2个字节保存到地址rt的通用寄存器的低2个字节。如图9-9所示。一个更加通用的描述如图9-10所示。

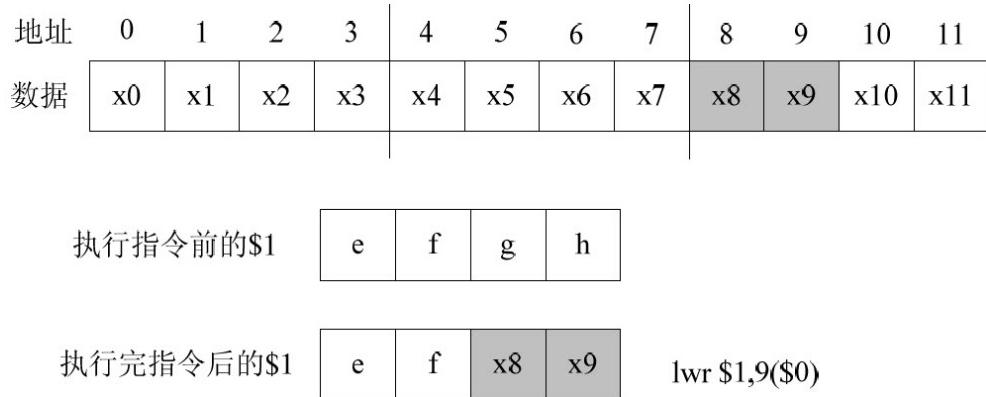


图9-9 lwr指令作用举例

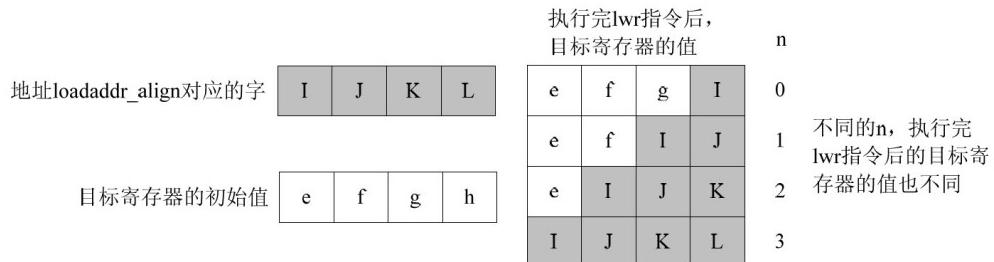


图9-10 lwr指令执行效果说明

lw与lwr指令配合可以实现从一个非对齐地址加载一个字，而且只需要使用2条指令，提高了效率。例如：使用一般指令从地址7处加载一个字，那么可以使用以下代码实现，共5条指令。

```

lw  $1, 4($0)          # 取得地址0x4处的字，保存在$1中
lw  $2, 8($0)          # 取得地址0x8处的字，保存在$2中
sll $1, $1, 24          # $1左移24位
slr $2, $2, 8           # $2右移8位
or   $1, $1, $2          # $1与$2进行逻辑“或”运算，得到最终结果

```

而有了lwl、lwr指令后，只需要2条指令即可。如下，图9-11是对这个过程的描述。

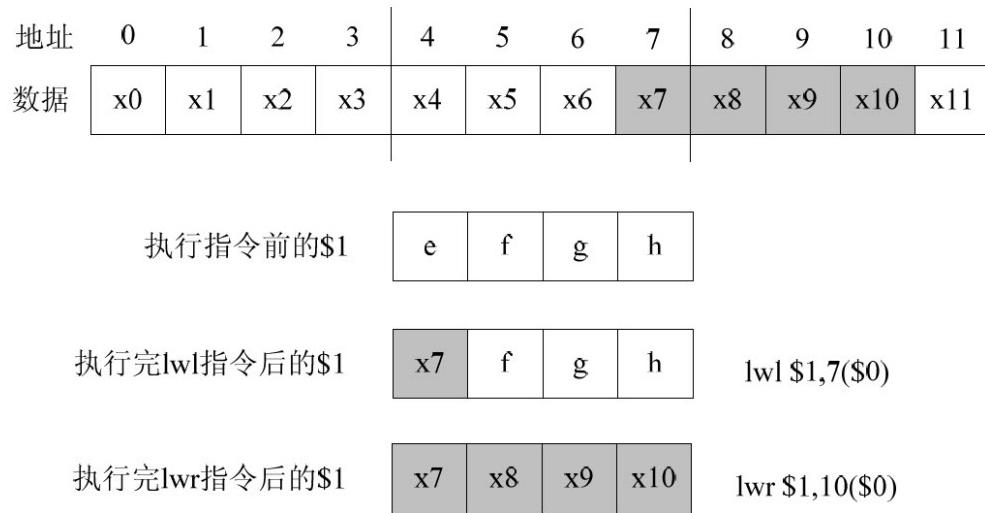


图9-11 组合使用指令lwl、lwr，可以加载非对齐地址的字

```
lwl $1, 7($0)
```

```
lwr $1,10($0)
```

9.1.5 存储指令swl、swr说明

存储指令swl、swr的格式如图9-12所示。

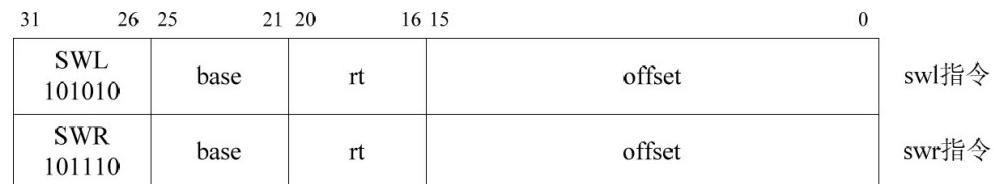


图9-12 存储指令swl、swr的格式

- 当指令中的指令码为6'b101010时，是swl指令，非对齐存储指令，向左存储。

指令用法为：swl rt, offset(base)。

指令作用为：将地址为rt的通用寄存器的高位部分存储到内存中指定的地址处，存储地址的最后两位确定了要存储rt通用寄存器的哪几个字节。swl指令对存储地址没有对齐要求，这是与前面介绍的sh、sw指令的不同之处。在大端模式、小端模式下，swl指令的效果不同，因为OpenMIPS是大端模式，所以此处只介绍在大端模式下swl指令的效果。假设计算出来的存储地址是storeaddr，storeaddr最低两位的值为n，storeaddr最低两位设为0后的值称为storeaddr_align，如下。

存储地址storeaddr = signed_extended(offset) + GPR[base]

n = storeaddr[1:0]

storeaddr_align = storeaddr - n

例如：假设计算出来的存储地址是5，swl指令要向地址5存储数据，那么storeaddr就等于5，n等于1，storeaddr_align等于4。

swl指令的作用是将地址为rt的通用寄存器的最高 $4-n$ 个字节存储到地址storeaddr处。

继续上例，此时storeaddr_align为4，n为1，所以将地址rt的通用寄存器的最高3个字节存储到从地址5开始处，对应的是地址为5、6、7的三个字节，如图9-13所示。一个更加通用的描述如图9-14所示。

通用寄存器\$1的值

e	f	g	h
---	---	---	---

执行swl指令之前的内存

地址	0	1	2	3	4	5	6	7	8	9	10	11
数据	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11

swl \$1,5(\$0)

执行swl指令之后的内存

地址	0	1	2	3	4	5	6	7	8	9	10	11
数据	x0	x1	x2	x3	x4	e	f	g	x8	x9	x10	x11

图9-13 swl指令作用举例

内存地址storeaddr_align对应的字	执行完swl指令后，内存地址storeaddr_align对应内容				n	
	I	J	K	L		
	e	f	g	h	0	不同的n，执行完swl指令后的内存地址storeaddr_align处存储的值也不同
	I	e	f	g	1	swl指令后的内存地址storeaddr_align处
	I	J	e	f	2	存储的值也不同
	I	J	K	e	3	

图9-14 swl指令执行效果说明

- 当指令中的指令码为6'b101110时，是swr指令，非对齐存储指令，向右存储。

指令用法为：swr rt, offset(base)。

指令作用为：将地址为rt的通用寄存器的低位部分存储到内存中指定的地址处，存储地址的最后两位确定了要存储rt通用寄存器的哪几个字节。还是假设计算出来的存储地址是storeaddr，storeaddr的最低两位的值为n，storeaddr最低两位设为0后的值称为storeaddr_align，如下。

存储地址 $\text{storeaddr} = \text{signed_extended}(\text{offset}) + \text{GPR}[\text{base}]$

n = storeaddr[1:0]

storeaddr_align = storeaddr - n

例如：假设计算出来的存储地址是9，swr指令要向地址9存储数据，那么storeaddr就等于9，n等于1，storeaddr_align等于8。

swr指令的作用是将地址为rt的通用寄存器的最低n+1个字节存储到地址storeaddr_align处。

继续上例，此时storeaddr_align为8，n为1，所以将地址rt的通用寄存器的最低2个字节存储到从地址8开始处，对应的是地址为8、9的两个位置，如图9-15所示。一个更加通用的描述如图9-16所示。

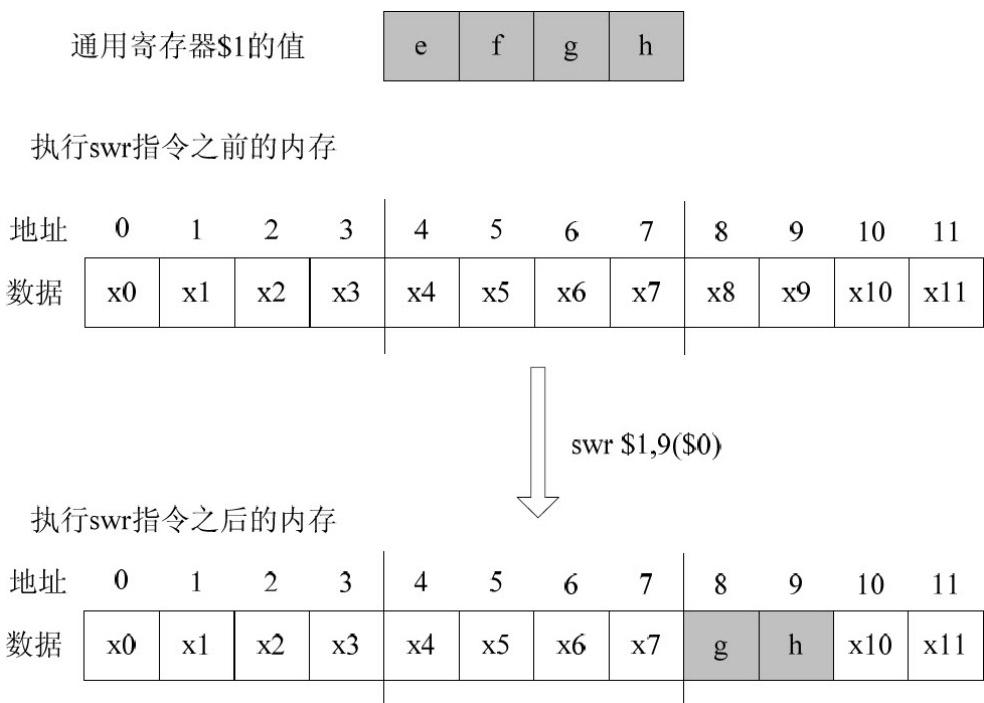


图9-15 swr指令作用举例

执行完swr指令后，内存地址storeaddr_align对应内容				n
内存地址storeaddr_align对应的字	I	J	K	L
地址为rt的通用寄存器的值	e	f	g	h
	h	J	K	L
	g	h	K	L
	f	g	h	L
	e	f	g	h

图9-16 swr指令执行效果说明

swl与swr指令配合可以实现向一个非对齐地址存储一个字，而且只须使用2条指令，提高了效率。例如：使用一般指令向地址7处存储一个字，那么可以使用以下代码实现，共5条指令。

```
sll $2, $1, 24      # 要存储的数据在$1中，将$1的最高字节存储到$2
sb  $2, 7($0)       # 存储最高字节到地址为7的内存处
sll $2, $1, 8        # 将$1的第2、1字节保存到$2中
sh  $2, 8($0)       # 存储第2、1字节到地址为8、9的内存处
sb  $1, 10($0)       # 存储第0字节到地址为10的内存处
```

而有了swl、swr指令后，只需要2条指令即可。如下，图9-17是对这个过程的描述。

通用寄存器\$1的值

e	f	g	h
---	---	---	---

执行swr指令之前的内存

地址	0	1	2	3	4	5	6	7	8	9	10	11
数据	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11

swl \$1,7(\$0)

执行swl指令之后的内存

地址	0	1	2	3	4	5	6	7	8	9	10	11
数据	x0	x1	x2	x3	x4	x5	x6	e	x8	x9	x10	x11

swr \$1,10(\$0)

执行swr指令之后的内存

地址	0	1	2	3	4	5	6	7	8	9	10	11
数据	x0	x1	x2	x3	x4	x5	x6	e	f	g	h	x11

图9-17 组合使用指令swl、swr，可以向非对齐地址存储字

```
swl $1, 7($0)
```

```
swr $1, 10($0)
```

9.2 加载存储指令实现思路

本节介绍除ll、sc之外的加载存储指令的实现思路，ll、sc指令的实现思路将在9.7节专题介绍。

1. 加载指令实现思路

加载指令在译码阶段进行译码，得到运算类型aluse_l_o、aluop_o，以及要写的目的寄存器信息。这些信息传递到执行阶段，然后又传递到访存阶段，访存阶段依据这些信息，设置对数据存储器RAM的访问信号。从RAM读取回来的数据需要按照加载指令的类型、加载地址进行对齐调整，调整后的结果作为最终要写入目的寄存器的数据。

2. 存储指令实现思路

存储指令在译码阶段进行译码，得到运算类型aluse_l_o、aluop_o，以及要存储的数据。这些信息传递到执行阶段，然后又传递到访存阶段，访存阶段依据这些信息，设置对数据存储器RAM的访问信号，将数据写入RAM。

需要特别注意的是：本章假设可以在一个时钟周期内完成对外部数据存储器RAM的读、写操作，在后续章节实现实践版OpenMIPS处理器的时候会考虑复杂情况。

9.2.1 数据流图的修改

为了实现除ll、sc之外的加载存储指令，修改数据流图如图9-18所示。主要是在访存阶段增加了对数据存储器RAM的访问，同时，由于要写入目的寄存器的数据可能是执行阶段的结果，也可能是在访存阶段从数据存储器RAM加载得到的数据，所以在访存阶段增加了一个多路选择器，进行选择。

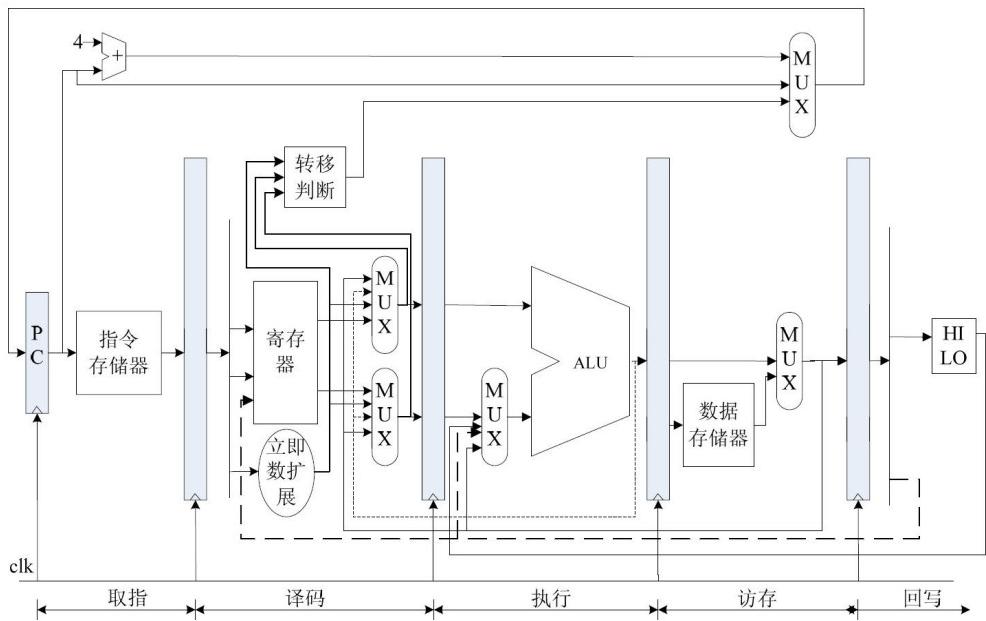


图9-18 为了实现除ll、sc之外的加载存储指令而修改的数据流图

9.2.2 系统结构的修改

为了实现除ll、sc之外的加载存储指令，需要对系统结构进行修改，增加部分模块的接口，修改后的系统结构如图9-19所示。

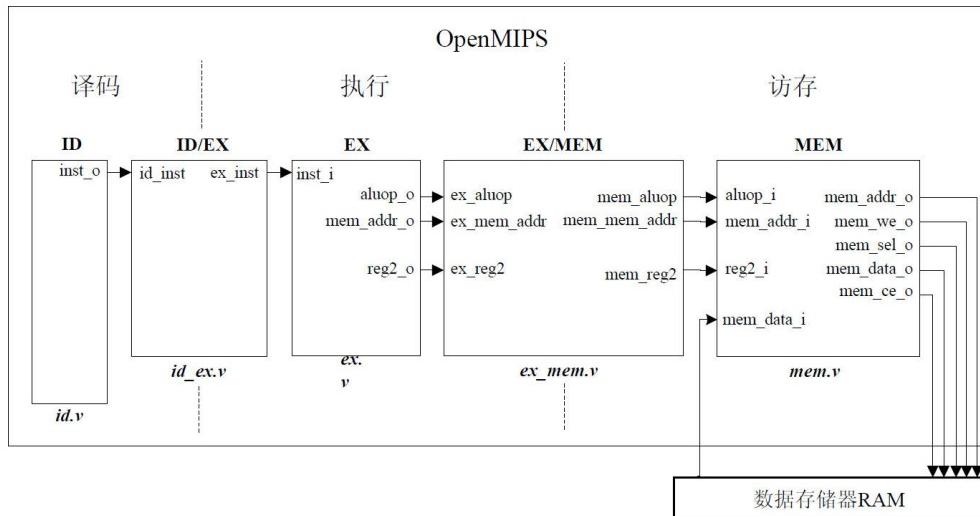


图9-19 为了实现除ll、sc之外的加载存储指令而对系统结构的修改

主要修改内容如下。

- (1) 译码阶段的ID模块增加了输出信号inst_o，其值就是处于译码阶段的指令，该信号会传递到执行阶段，在执行阶段的EX模块会利用该信号的值计算加载、存储地址mem_addr_o。
- (2) 执行阶段的EX模块将运算子类型aluop_o、加载存储地址mem_addr_o、读取的第二个操作数reg2_o等信息，通过EX/MEM模块传递到访存阶段的MEM模块。
- (3) 访存阶段的MEM模块依据加载、存储指令的类型，确定对数据存储器RAM的访问信息，通过mem_ce_o接口送出数据存储器使能信号，mem_addr_o接口送出访问地址，mem_we_o接口指出是加载还是存储操作、mem_sel_o接口送出字节选择信号，如果是存储指令，那么还通过mem_data_o接口输出要存储的数据，如果是加载指令，那么会从mem_data_i接口获得读取到的数据，然后MEM模块依据具体的加载指令类型、加载地址，对获取的数据进行对齐调整，最终得到要写入目的寄存器的数据。

9.3 修改OpenMIPS以实现加载存储指令

9.3.1 修改译码阶段

1. 修改ID模块

参考图9-19可知，ID模块要增加接口inst_o，如表9-1所示。

表9-1 ID模块新增加的接口描述

序号	接口名	宽度 (bit)	输入/输出	作用
1	inst_o	32	输出	当前处于译码阶段的指令

在ID模块还要增加对加载存储指令的分析，根据图9-1、图9-2、图9-6、图9-12给出的加载存储指令的格式可知，这些指令的指令码都是不同的，所以可以直接依据指令码确定是哪一种指令，确定指令的过程如图9-20所示。

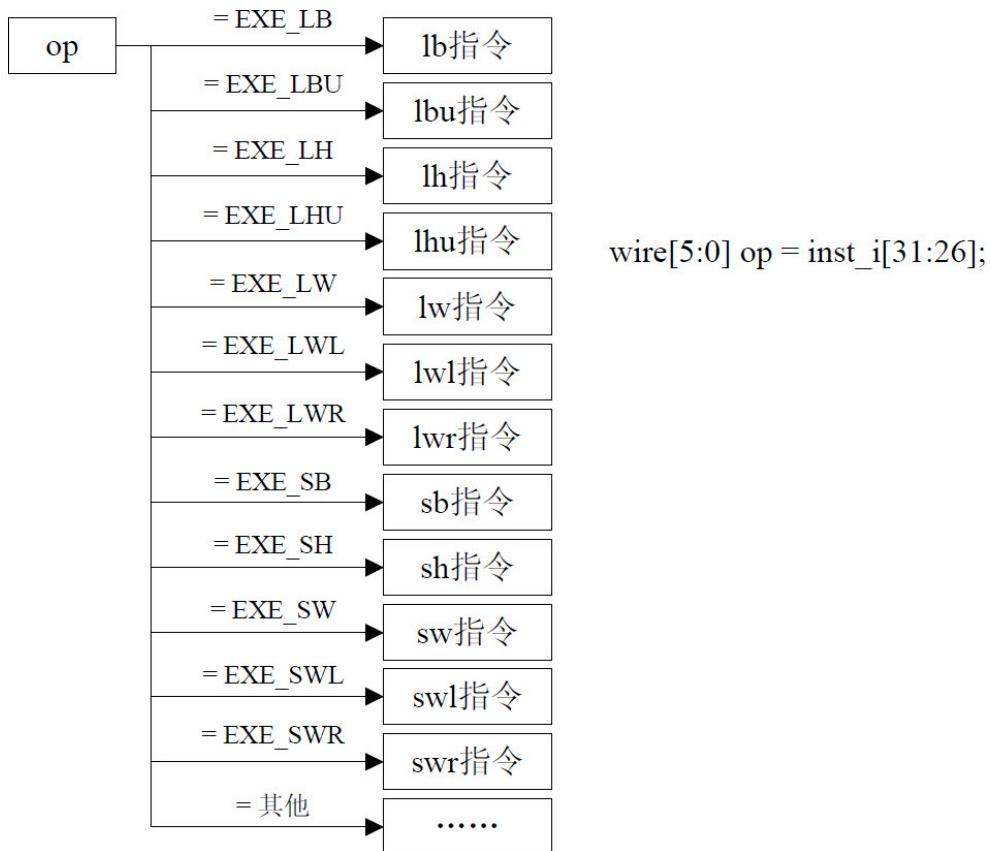


图9-20 确定加载存储指令的过程

其中涉及的宏定义如下，正是各个加载存储指令的指令码，在本书附带光盘Code\Chapter9_1目录下的defines.v文件可以找到这些定义。

```
`define EXE_LB    6'b100000
`define EXE_LBU   6'b100100
`define EXE_LH    6'b100001
`define EXE_LHU   6'b100101
`define EXE_LW    6'b100011
`define EXE_LWL   6'b100010
`define EXE_LWR   6'b100110
`define EXE_SB    6'b101000
`define EXE_SH    6'b101001
`define EXE_SW    6'b101011
`define EXE_SWL   6'b101010
`define EXE_SWR   6'b101110
```

修改译码阶段的ID模块如下。完整代码请参考本书附带光盘中Code\Chapter9_1目录下的id.v文件。

```
module id(
    .....
    output wire[`RegBus]           inst_o,      // 新增加的输出接口
    .....
);
    .....
```

```
assign inst_o = inst_i; // inst_o的值就是译码阶段的指令

always @ (*) begin
    if (rst == `RstEnable) begin
        .....
    end else begin
        aluop_o      <= `EXE_NOP_OP;
        alusel_o     <= `EXE_RES_NOP;
        wd_o          <= inst_i[15:11]; // 默认目的寄存器
地址wd_o
        wreg_o       <= `WriteDisable;
        instvalid    <= `InstInvalid;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= inst_i[25:21]; // 默认的
reg1_addr_o
        reg2_addr_o <= inst_i[20:16]; // 默认的
reg2_addr_o
        imm          <= `ZeroWord;
        .....
        case (op)
            .....
`EXE_LB:   begin // 1b指令
```

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_LB_OP;
alusel_o     <= `EXE_RES_LOAD_STORE;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
wd_o         <= inst_i[20:16];
instinvalid <= `InstInvalid;
end
`EXE_LBU:   begin           // lbu指令
```

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_LBU_OP;
alusel_o     <= `EXE_RES_LOAD_STORE;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
wd_o         <= inst_i[20:16];
instinvalid <= `InstInvalid;
end
`EXE_LH:    begin           // lh指令
```

```
wreg_o      <= `WriteEnable;
aluop_o      <= `EXE_LH_OP;
alusel_o     <= `EXE_RES_LOAD_STORE;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
```

```
        wd_o          <= inst_i[20:16];
        instinvalid  <= `InstInvalid;
    end

`EXE_LHU:   begin           // lhu指令

        wreg_o       <= `WriteEnable;
        aluop_o      <= `EXE_LHU_OP;
        alusel_o     <= `EXE_RES_LOAD_STORE;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        wd_o          <= inst_i[20:16];
        instinvalid  <= `InstInvalid;
    end

`EXE_LW:    begin           // lw指令

        wreg_o       <= `WriteEnable;
        aluop_o      <= `EXE_LW_OP;
        alusel_o     <= `EXE_RES_LOAD_STORE;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        wd_o          <= inst_i[20:16];
        instinvalid  <= `InstInvalid;
    end

`EXE_LWL:   begin           // lwl指令
```

```
wreg_o      <= `WriteEnable;
aluop_o     <= `EXE_LWL_OP;
alusel_o    <= `EXE_RES_LOAD_STORE;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
wd_o        <= inst_i[20:16];
instinvalid <= `InstInvalid;
end
```

`EXE_LWR: *begin* // lwr指令

```
wreg_o      <= `WriteEnable;
aluop_o     <= `EXE_LWR_OP;
alusel_o    <= `EXE_RES_LOAD_STORE;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
wd_o        <= inst_i[20:16];
instinvalid <= `InstInvalid;
end
```

`EXE_SB: *begin* // sb指令

```
wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_SB_OP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
```

```

    instvalid    <= `InstValid;
    alusel_o     <= `EXE_RES_LOAD_STORE;
end

`EXE_SH:      begin          // sh指令

    wreg_o       <= `WriteDisable;
    aluop_o      <= `EXE_SH_OP;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid    <= `InstValid;
    alusel_o     <= `EXE_RES_LOAD_STORE;
end

`EXE_SW:      begin          // sw指令

    wreg_o       <= `WriteDisable;
    aluop_o      <= `EXE_SW_OP;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instvalid    <= `InstValid;
    alusel_o     <= `EXE_RES_LOAD_STORE;
end

`EXE_SWL:     begin         // swl指令

    wreg_o       <= `WriteDisable;

```

```

    aluop_o      <= `EXE_SWL_OP;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instinvalid <= `InstInvalid;
    alusel_o     <= `EXE_RES_LOAD_STORE;
end

`EXE_SWR: begin // swr指令

    wreg_o      <= `WriteDisable;
    aluop_o      <= `EXE_SWR_OP;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    instinvalid <= `InstInvalid;
    alusel_o     <= `EXE_RES_LOAD_STORE;
end

.....

```

译码工作主要是确定要写的目的寄存器、要读取的寄存器和要执行的运算三个方面。以下对几个有代表性的指令的译码过程进行说明。

(1) lb指令

- 要写的目的寄存器：加载指令lb需要将加载结果写入目的寄存器，所以设置wreg_o为WriteEnable，同时参考图9-1可知，要写的目的寄存器地址是指令中的第16~20bit，所以设置wd_o为inst_i[20:16]。

- 要读取的寄存器：参考图9-1可知，计算加载目标地址需要使用到地址为base的寄存器值，所以设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器的值，默认读取的寄存器地址reg1_addr_o是指令的第21~25bit，正是lb指令中的base。所以最终译码阶段的输出reg1_o就是地址为base的寄存器的值。
- 要执行的运算：设置alusel_o为EXE_RES_LOAD_STORE，表示运算类型是加载存储，设置aluop_o为EXE_LB_OP，表示运算子类型是字节加载lb。

lbu、lh、lhu、lw指令与lb指令的译码过程类似，只是aluop_o的值不同。

(2) lwl指令

- 要写的目的寄存器：加载指令lwl需要将加载结果写入目的寄存器，所以设置wreg_o为WriteEnable，同时参考图9-6可知，要写的目的寄存器地址是指令中的第16~20bit，所以设置wd_o为inst_i[20:16]。
- 要读取的寄存器：参考图9-6可知，计算加载目标地址需要使用到地址为base的寄存器值，所以设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器的值，默认读取的寄存器地址reg1_addr_o是指令的第21~25bit，正是lwl指令中的base。所以最终译码阶段的输出reg1_o就是地址为base的寄存器的值。此外，由于lwl指令只是部分地修改目的寄存器，所以还需要读出目的寄存器，与lwl指令加载得到的结果进行组合，最终写入目的寄存器，因此，设置reg2_read_o也为1，表示通过Regfile模块的读端口2读取寄存器的值，默认读取的寄

存器地址reg2_addr_o是指令的第16~20bit，正是lw1指令中的rt。所以最终译码阶段的输出reg2_o就是地址为rt的寄存器的值。

- 要执行的运算：设置alusel_o为EXE_RES_LOAD_STORE，表示运算类型是加载存储，设置aluop_o为EXE_LWL_OP，表示运算子类型是向左加载lw1。

lwr指令与lw1指令的译码过程类似，只是aluop_o的值不同。

(3) sb指令

- 要写的目的寄存器：存储指令sb不需要写通用寄存器，所以设置wreg_o为WriteDisable。
- 要读取的寄存器：参考图9-2可知，计算存储目标地址需要使用的地址为base的寄存器值，所以设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器的值，默认读取的寄存器地址reg1_addr_o是指令的第21~25bit，正是sb指令中的base。所以最终译码阶段的输出reg1_o就是地址为base的寄存器的值。要存储的值是通用寄存器的值，所以设置reg2_read_o为1，表示通过Regfile模块的读端口2读取寄存器的值，默认读取的寄存器地址reg2_addr_o是指令的第16~20bit，正是sb指令中的rt。所以最终译码阶段的输出reg2_o就是地址为rt的寄存器的值。
- 要执行的运算：设置alusel_o为EXE_RES_LOAD_STORE，表示运算类型是加载存储指令，设置aluop_o为EXE_SB_OP，表示运算子类型是字节存储sb。

sh、sw、swr、swl指令与sb指令的译码过程类似，只是aluop_o的值不同。

2. 修改ID/EX模块

参考图9-19可知，ID/EX模块需要增加部分接口，用于将ID模块新增加的输出信号inst_o传递到执行阶段的EX模块。如表9-2所示。

表9-2 ID/EX模块新增加的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	id_inst	32	输入	当前处于译码阶段的指令
2	ex_inst	32	输出	当前处于执行阶段的指令

修改译码阶段的ID/EX模块如下。完整代码位于本书附带光盘中Code\Chapter9_1目录下的id_ex.v文件中。

```
module id_ex(  
    .....  
    input wire[`RegBus]           id_inst, // 来自ID模块的信号  
    .....  
    output reg[`RegBus]          ex_inst // 传递到EX模块
```

```
);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        .....
        ex_inst <= `ZeroWord;
    end else if(stall[2] == `Stop && stall[3] == `NoStop)
begin
    .....
    ex_inst <= `ZeroWord;
end else if(stall[2] == `NoStop) begin
    .....
    //在译码阶段没有暂停的情况下，直接将ID模块的输入通过接口
ex_inst输出
    ex_inst <= id_inst;
end
end
endmodule
```

9.3.2 修改执行阶段

1. 修改EX模块

在执行阶段的EX模块会计算加载存储的目的地址，参考图9-19可知，EX模块会增加部分接口，如表9-3所示。

表9-3 EX模块新增加接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	inst_i	32	输入	当前处于执行阶段的指令
2	aluop_o	8	输出	执行阶段的指令要进行的运算子类型
3	mem_addr_o	32	输出	加载、存储指令对应的存储器地址
4	reg2_o	32	输出	存储指令要存储的数据，或者 lwl、lwr 指令要加载到的目的寄存器的原始值

修改执行阶段的EX模块如下。完整代码位于本书附带光盘中 Code\Chapter9_1 目录下的 ex.v 文件中。

```
module ex(  
    .....  
    //新增输入接口inst_i，其值就是当前处于执行阶段的指令  
    input wire[`RegBus] inst_i,  
    .....  
    //下面新增的几个输出接口是为加载、存储指令准备的  
    output wire[`AluOpBus] aluop_o,  
    output wire[`RegBus] mem_addr_o,  
    output wire[`RegBus] reg2_o,  
    .....  
);
```

```
.....  
  
//aluop_o会传递到访存阶段，届时将利用其确定加载、存储类型  
assign aluop_o = aluop_i;  
  
//mem_addr_o会传递到访存阶段，是加载、存储指令对应的存储器地址，此  
处的reg1_i  
//就是加载、存储指令中地址为base的通用寄存器的值，inst_i[15:0]就是  
指令中的  
//offset。通过mem_addr_o的计算，读者也可以明白为何要在译码阶段ID模  
块新增输入  
//出接口inst_o  
assign mem_addr_o = reg1_i +  
{16{inst_i[15]}}, inst_i[15:0];  
  
//reg2_i是存储指令要存储的数据，或者lw1、lwr指令要加载到的目的寄存  
器的原始值，  
//将该值通过reg2_o接口传递到访存阶段  
assign reg2_o = reg2_i;  
.....
```

2. 修改EX/MEM模块

参考图9-19可知，EX/MEM模块会增加部分接口，用于将EX模块新增的输出传递到访存阶段，增加的接口描述如表9-4所示。

表9-4 EX/MEM模块新增加接口的描述

序号	接 口 名	宽度(bit)	输入/输出	作 用
1	ex_aluop	8	输入	执行阶段的指令要进行的运算的子类型
2	ex_mem_addr	32	输入	执行阶段的加载、存储指令对应的存储器地址
3	ex_reg2	32	输入	执行阶段的存储指令要存储的数据，或者lw1、lwr指令要写入的目的寄存器的原始值
4	mem_aluop	8	输出	访存阶段的指令要进行的运算的子类型
5	mem_mem_addr	32	输出	访存阶段的加载、存储指令对应的存储器地址
6	mem_reg2	32	输出	访存阶段的存储指令要存储的数据，或者lw1、lwr指令要写入的目的寄存器的原始值

修改执行阶段的EX/MEM模块如下，只是一个简单的传递操作，当流水线的执行阶段没有被暂停时，将来自执行阶段EX模块的输出传递到访存阶段。完整代码请参考本书附带光盘Code\Chapter9_1目录下的ex_mem.v文件。

```
module ex_mem(
    .....  

    //为实现加载、存储指令而添加的输入接口  

    input wire[`AluOpBus]           ex_aluop,  

    input wire[`RegBus]             ex_mem_addr,  

    input wire[`RegBus]             ex_reg2,  

    .....  

    //为实现加载、存储指令而添加的输出接口  

    output reg[`AluOpBus]          mem_aluop,  

    output reg[`RegBus]            mem_mem_addr,
```

```
output reg[`RegBus]           mem_reg2,  
.....  
);  
  
always @ (posedge clk) begin  
    if(rst == `RstEnable) begin  
        .....  
        mem_aluop     <= `EXE_NOP_OP;  
        mem_mem_addr <= `ZeroWord;  
        mem_reg2      <= `ZeroWord;  
    end else if(stall[3] == `Stop && stall[4] == `NoStop)  
begin  
    .....  
    mem_aluop     <= `EXE_NOP_OP;  
    mem_mem_addr <= `ZeroWord;  
    mem_reg2      <= `ZeroWord;  
end else if(stall[3] == `NoStop) begin
```

```

      .....
      mem_aluop     <= ex_aluop;
      mem_mem_addr <= ex_mem_addr;
      mem_reg2      <= ex_reg2;
    end else begin
      .....
    end
  end

endmodule

```

9.3.3 修改访存阶段

访存阶段主要是修改MEM模块，参考图9-19可知，需要为其添加对数据存储器RAM的访问接口，添加的接口描述如表9-5所示。

表9-5 MEM模块新增加接口的描述

序号	接口名	宽度(bit)	输入/输出	作用
1	aluop_i	8	输入	访存阶段的指令要进行的运算的子类型
2	mem_addr_i	32	输入	访存阶段的加载、存储指令对应的存储器地址

续表

序号	接口名	宽度(bit)	输入/输出	作用
3	reg2_i	32	输入	访存阶段的存储指令要存储的数据，或者lw1、lwr指令要写入的目的寄存器的原始值
4	mem_data_i	32	输入	从数据存储器读取的数据
5	mem_addr_o	32	输出	要访问的数据存储器的地址
6	mem_we_o	1	输出	是否是写操作，为1表示是写操作
7	mem_sel_o	4	输出	字节选择信号
8	mem_data_o	32	输出	要写入数据存储器的数据
9	mem_ce_o	1	输出	数据存储器使能信号

从图9-19可知，表9-5后面的几个新增接口mem_data_i、mem_addr_o、mem_we_o、mem_sel_o、mem_data_o、mem_ce_o都是与数据存储器相连的，其中大部分接口的作用都很好理解，此处只对mem_sel_o做进一步说明，分加载、存储两种操作分别说明。

(1) 对于加载操作，MIPS32指令集架构中定义的加载指令可以加载字节、半字、字，但是数据总线的宽度是32位，占4个字节。如果执行加载字节指令lb、lbu，那么就要知道通过数据总线输入的4个字节中，哪个字节是要读取的数据；如果执行加载半字指令lh、lhu，那么就要知道哪个半字是要读取的数据，mem_sel_o的作用就是指出哪一部分是有效数据。mem_sel_o宽度为4，分别对应数据总线的4个字节，比如：使用加载指令lb读取数据存储器地址0x1处的字节，那么可以设置mem_sel_o为4'b0100，意思就是，希望外部存储器在输出数据时，将地址0x1处的字节放在32位数据总线的次高字节，也就是第16~23bit的位置，当数据送到处理器时，处理器就取出其中第16~23bit对应的字节，作为数据存储器地址0x1处的值。

(2) 对于存储操作，MIPS32指令集架构中定义的存储指令可以存储字节、半字、字，但是数据总线的宽度是32位，占4个字节，如果执行字节存储指令sb、半字存储指令sh，那么外部数据存储器就要知道通过数据总线传递过来的4个字节中，哪个字节、哪个半字是要存储的数据，mem_sel_o的作用就是指出哪一部分是要存储的有效数据。比如：使用存储指令sh向地址0x2处存储0x8281，那么可以设置mem_data_o为0x82818281、设置mem_sel_o为4'b0011，这样外部存储器就知道要存储的数据是0x82818281的最低两个字节，正是0x8281。

访存阶段MEM模块的代码主要修改如下，完整代码请读者参考本书附带光盘中Code\Chapter9_1目录下的mem.v文件。

```
module mem(  
    .....  
  
    //新增接口，来自执行阶段的信息  
    input wire[`AluOpBus] aluop_i,  
    input wire[`RegBus] mem_addr_i,  
    input wire[`RegBus] reg2_i,  
  
    //新增接口，来自外部数据存储器RAM的信息  
    input wire[`RegBus] mem_data_i,  
    .....  
  
    //新增接口，送到外部数据存储器RAM的信息  
    output reg[`RegBus] mem_addr_o,  
    output wire mem_we_o,  
    output reg[3:0] mem_sel_o,  
    output reg[`RegBus] mem_data_o,  
    output reg mem_ce_o  
  
);  
  
    wire[`RegBus] zero32;  
    reg mem_we;  
  
    assign mem_we_o = mem_we; //外部数据存储器RAM的读、写信号
```

```

assign zero32 = `Zeroword;

always @ (*) begin
    if(rst == `RstEnable) begin
        wd_o      <= `NOPRegAddr;
        wreg_o    <= `WriteDisable;
        wdata_o   <= `ZeroWord;
        hi_o      <= `ZeroWord;
        lo_o      <= `ZeroWord;
        whilo_o   <= `WriteDisable;
        mem_addr_o <= `ZeroWord;
        mem_we     <= `WriteDisable;
        mem_sel_o  <= 4'b0000;
        mem_data_o <= `ZeroWord;
        mem_ce_o   <= `ChipDisable;
    end else begin
        wd_o      <= wd_i;
        wreg_o    <= wreg_i;
        wdata_o   <= wdata_i;
        hi_o      <= hi_i;
        lo_o      <= lo_i;
        whilo_o   <= whilo_i;
        mem_we     <= `WriteDisable;
        mem_addr_o <= `ZeroWord;
        mem_sel_o  <= 4'b1111;
        mem_ce_o   <= `ChipDisable;
    case (aluop_i)

```

```

`EXE_LB_OP: begin //lb指令

    mem_addr_o <= mem_addr_i;
    mem_we      <= `WriteDisable;
    mem_ce_o    <= `ChipEnable;

    case (mem_addr_i[1:0])
        2'b00: begin
            wdata_o <=
{{24{mem_data_i[31]}},mem_data_i[31:24]};

            mem_sel_o <= 4'b1000;
        end
        2'b01: begin
            wdata_o <=
{{24{mem_data_i[23]}},mem_data_i[23:16]};

            mem_sel_o <= 4'b0100;
        end
        2'b10: begin
            wdata_o <=
{{24{mem_data_i[15]}},mem_data_i[15:8]};

            mem_sel_o <= 4'b0010;
        end
        2'b11: begin
            wdata_o <=
{{24{mem_data_i[7]}},mem_data_i[7:0]};

            mem_sel_o <= 4'b0001;
        end
    end

```

```

    default: begin
        wdata_o     <= `ZeroWord;
    end
endcase
end

`EXE_LBU_OP: begin //lbu指令

mem_addr_o <= mem_addr_i;
mem_we      <= `WriteDisable;
mem_ce_o    <= `ChipEnable;
case (mem_addr_i[1:0])
    2'b00: begin
        wdata_o     <= {{24{1'b0}},mem_data_i[31:24]};
        mem_sel_o  <= 4'b1000;
    end
    2'b01: begin
        wdata_o     <= {{24{1'b0}},mem_data_i[23:16]};
        mem_sel_o  <= 4'b0100;
    end
    2'b10:      begin
        wdata_o     <= {{24{1'b0}},mem_data_i[15:8]};
        mem_sel_o  <= 4'b0010;
    end
    2'b11: begin
        wdata_o     <= {{24{1'b0}},mem_data_i[7:0]};
        mem_sel_o  <= 4'b0001;
    end

```

```

        end

        default: begin
            wdata_o    <= `ZeroWord;
        end
    endcase
end

`EXE_LH_OP: begin //lh指令

mem_addr_o <= mem_addr_i;
mem_we     <= `WriteDisable;
mem_ce_o   <= `ChipEnable;
case (mem_addr_i[1:0])
    2'b00: begin
        wdata_o      <=
{{16{mem_data_i[31]}},mem_data_i[31:16]};

        mem_sel_o <= 4'b1100;
    end
    2'b10: begin
        wdata_o      <=
{{16{mem_data_i[15]}},mem_data_i[15:0]};

        mem_sel_o <= 4'b0011;
    end
    default: begin
        wdata_o    <= `ZeroWord;
    end
endcase

```

```

    end

`EXE_LHU_OP: begin //lhu指令

    mem_addr_o <= mem_addr_i;
    mem_we      <= `WriteDisable;
    mem_ce_o    <= `ChipEnable;

    case (mem_addr_i[1:0])
        2'b00: begin
            wdata_o   <= {{16{1'b0}},mem_data_i[31:16]};
            mem_sel_o <= 4'b1100;
        end
        2'b10: begin
            wdata_o   <= {{16{1'b0}},mem_data_i[15:0]};
            mem_sel_o <= 4'b0011;
        end
        default: begin
            wdata_o   <= `ZeroWord;
        end
    endcase
end

`EXE_LW_OP: begin //lw指令

```

```

    mem_addr_o <= mem_addr_i;
    mem_we      <= `WriteDisable;
    wdata_o     <= mem_data_i;

```

```

    mem_sel_o  <= 4'b1111;
    mem_ce_o   <= `ChipEnable;
end

`EXE_LWL_OP: begin //lw1指令

    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we     <= `WriteDisable;
    mem_sel_o  <= 4'b1111;
    mem_ce_o   <= `ChipEnable;
    case (mem_addr_i[1:0])
        2'b00: begin
            wdata_o <= mem_data_i[31:0];
        end
        2'b01: begin
            wdata_o <= {mem_data_i[23:0], reg2_i[7:0]};
        end
        2'b10: begin
            wdata_o <= {mem_data_i[15:0], reg2_i[15:0]};
        end
        2'b11: begin
            wdata_o <= {mem_data_i[7:0], reg2_i[23:0]};
        end
        default: begin
            wdata_o <= `ZeroWord;
        end
    endcase

```

```

    end

`EXE_LWR_OP: begin //lwr指令

    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we      <= `WriteDisable;
    mem_sel_o   <= 4'b1111;
    mem_ce_o    <= `ChipEnable;
    case (mem_addr_i[1:0])
        2'b00: begin
            wdata_o <= {reg2_i[31:8],mem_data_i[31:24]};
        end
        2'b01: begin
            wdata_o <= {reg2_i[31:16],mem_data_i[31:16]};
        end
        2'b10: begin
            wdata_o <= {reg2_i[31:24],mem_data_i[31:8]};
        end
        2'b11: begin
            wdata_o <= mem_data_i;
        end
        default: begin
            wdata_o <= `ZeroWord;
        end
    endcase
end

`EXE_SB_OP: begin //sb指令

```

```
mem_addr_o <= mem_addr_i;
mem_we      <= `WriteEnable;
mem_data_o <= {reg2_i[7:0],reg2_i[7:0],
                reg2_i[7:0],reg2_i[7:0]};
mem_ce_o    <= `ChipEnable;
case (mem_addr_i[1:0])
  2'b00: begin
    mem_sel_o <= 4'b1000;
  end
  2'b01: begin
    mem_sel_o <= 4'b0100;
  end
  2'b10: begin
    mem_sel_o <= 4'b0010;
  end
  2'b11: begin
    mem_sel_o <= 4'b0001;
  end
  default: begin
    mem_sel_o <= 4'b0000;
  end
endcase
end
`EXE_SH_OP: begin //sh指令
```

```
mem_addr_o <= mem_addr_i;  
mem_we      <= `WriteEnable;  
mem_data_o <= {reg2_i[15:0], reg2_i[15:0]};  
mem_ce_o   <= `ChipEnable;  
case (mem_addr_i[1:0])  
    2'b00: begin  
        mem_sel_o <= 4'b1100;  
    end  
    2'b10: begin  
        mem_sel_o <= 4'b0011;  
    end  
    default: begin  
        mem_sel_o <= 4'b0000;  
    end  
endcase  
end  
`EXE_SW_OP: begin //sw指令
```

```
mem_addr_o <= mem_addr_i;  
mem_we      <= `WriteEnable;  
mem_data_o <= reg2_i;  
mem_sel_o  <= 4'b1111;  
mem_ce_o   <= `ChipEnable;  
end  
`EXE_SWL_OP: begin //swl指令
```

```

mem_addr_o <= {mem_addr_i[31:2], 2'b00};

mem_we      <= `WriteEnable;
mem_ce_o    <= `ChipEnable;

case (mem_addr_i[1:0])
  2'b00: begin
    mem_sel_o <= 4'b1111;
    mem_data_o <= reg2_i;
  end
  2'b01: begin
    mem_sel_o <= 4'b0111;
    mem_data_o <= {zero32[7:0],reg2_i[31:8]};
  end
  2'b10: begin
    mem_sel_o <= 4'b0011;
    mem_data_o <= {zero32[15:0],reg2_i[31:16]};
  end
  2'b11: begin
    mem_sel_o <= 4'b0001;
    mem_data_o <= {zero32[23:0],reg2_i[31:24]};
  end
  default: begin
    mem_sel_o <= 4'b0000;
  end
endcase
end

```

```

`EXE_SWR_OP: begin //swr指令

    mem_addr_o <= {mem_addr_i[31:2], 2'b00};
    mem_we      <= `WriteEnable;
    mem_ce_o    <= `ChipEnable;

    case (mem_addr_i[1:0])
        2'b00: begin
            mem_sel_o <= 4'b1000;
            mem_data_o <= {reg2_i[7:0], zero32[23:0]};
        end
        2'b01: begin
            mem_sel_o <= 4'b1100;
            mem_data_o <= {reg2_i[15:0], zero32[15:0]};
        end
        2'b10: begin
            mem_sel_o <= 4'b1110;
            mem_data_o <= {reg2_i[23:0], zero32[7:0]};
        end
        2'b11: begin
            mem_sel_o <= 4'b1111;
            mem_data_o <= reg2_i[31:0];
        end
        default: begin
            mem_sel_o <= 4'b0000;
        end
    endcase

```

```
        end

        default: begin
            //do nothing
        end
    endcase
end

endmodule
```

上面的代码虽然很长，但结构很清晰，作用也很明确，就是依据不同的加载、存储指令类型，给出 mem_addr_o、mem_we_o、mem_sel_o、mem_data_o、wdata_o 等接口的值。下面对其中几个典型指令的访存过程进行解释。

1. lb指令的访存过程

- (1) 因为要访问数据存储器，所以设置mem_ce_o为ChipEnable。
- (2) 因为是加载操作，所以设置mem_we_o为WriteDisable。
- (3) 给出要访问的数据存储器地址mem_addr_o，其值就是执行阶段计算出来的地址mem_addr_i。
- (4) 依据mem_addr_i的最后两位，确定mem_sel_o的值，并据此从数据存储器的输入数据mem_data_i中获得要读取的字节，进行符号扩展。比如：如果mem_addr_i的最后两位是01，那么设置mem_sel_o为4'b0100，表示希望数据存储器给出的数据的第16~23bit就是要读取的

字节，也就是 $\text{mem_data_i}[23:16]$ ，将其最高位进行符号扩展，得到最终的结果 wdata_o ，作为要写入目标寄存器的数据，读者如果忘记了 wdata_o 的作用，可以参考4.2.7节ori指令实现过程访存阶段的说明。

有些读者可能会感到疑惑，为何不直接设置 mem_sel_o 为4'b0001，表示希望数据存储器给出的数据的第0~7bit就是要读取的字节，而不考虑 mem_addr_i 的最后两位为何值，这样不是更简单吗？的确，这样做是更简单了，但是这里确定 mem_sel_o 值的过程实际上参考了Wishbone总线的相关规范，为的是在后期给OpenMIPS添加Wishbone总线接口的时候容易一些。在本章，读者可以简单地认为：外部的数据存储器并没有依据 mem_addr_o 地址读取数据，而是将 mem_addr_o 地址的最后两位修改为0，依据修改后的地址读取数据，所以OpenMIPS需要依据 mem_addr_o 最后两位的值，确定要读取的字节。如图9-21所示。

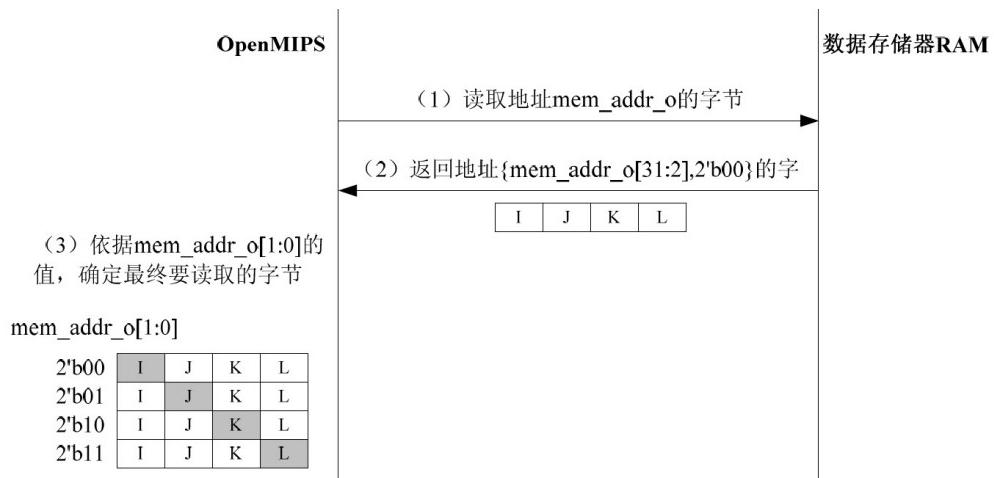


图9-21 lb指令的访存过程

lbu、lh、lhu、lw指令的访存过程与lb指令类似，可以对照理解。

2. lw指令的访存过程

- (1) 因为要访问数据存储器，所以设置mem_ce_o为ChipEnable。
- (2) 因为是加载操作，所以设置mem_we_o为WriteDisable。
- (3) 给出要访问的数据存储器地址mem_addr_o，其值就是执行阶段计算出来的地址mem_addr_i，但最后两位要设置为0，因为lwl指令要从RAM中读出一个字，所以需要将地址对齐，同时设置mem_sel_o为4'b1111。
- (4) 依据mem_addr_i的最后两位，将从数据存储器读取的数据mem_data_i与目的寄存器的原始值reg2_i进行组合，得到最终要写入目的寄存器的值wdata_o，组合过程可以参考图9-8。

lwr指令的访存过程与lwl指令类似，可以对照理解。

3. sb指令的访存过程

- (1) 因为要访问数据存储器，所以设置mem_ce_o为ChipEnable。
- (2) 因为是存储操作，所以设置mem_we_o为WriteEnable。
- (3) 给出要访问的数据存储器地址mem_addr_o，其值就是执行阶段计算出来的地址mem_addr_i。
- (4) sb指令要写入的数据是寄存器的最低字节，将该字节复制到mem_data_o的其余部分，然后依据mem_addr_i的最后两位，确定mem_sel_o的值。比如：如果mem_addr_i的最后两位是01，那么设置mem_sel_o为4'b0100，表示第16～23bit就是要写入的字节，即mem_data_o[23:16]。

读者可能又会有疑惑，为何不直接设置mem_sel_o的值为4'b0001，而不用考虑mem_addr_o最低两位的值，不用将最低字节复制到mem_data_o的其余部分呢？理由与lb指令一样，此处确定mem_sel_o值的过程实际上参考了Wishbone总线的相关规范，为的是后期给OpenMIPS添加Wishbone总线接口的时候容易一些。现在，大家可以简单地认为，外部的数据存储器并没有依据mem_addr_o存储数据，而是将mem_addr_o的最后两位修改为0，依据修改后的地址存储数据，所以OpenMIPS需要依据mem_addr_o最后两位的值，确定mem_sel_o的值，同时将最低字节复制到mem_data_o的其余部分，这样保证无论mem_sel_o为何值，写入字节始终是寄存器的最低字节。如图9-22所示。

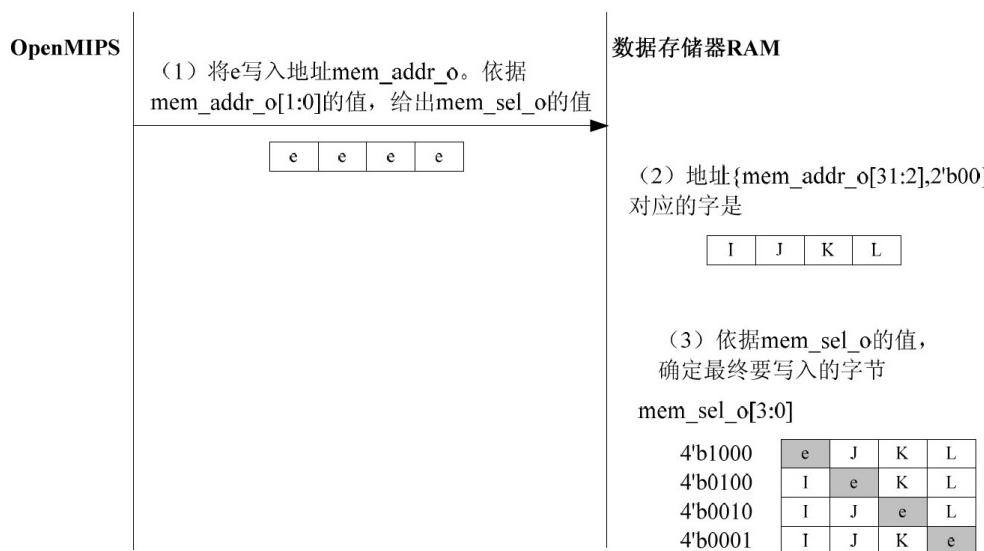


图9-22 sb指令的访存过程

sh、sw指令的访存过程与sb指令类似，可以对照理解。

4. swl指令的访存过程

(1) 要访问数据存储器，所以设置mem_ce_o为ChipEnable。

(2) 因为是存储操作，所以设置mem_we_o为WriteEnable。

(3) 给出要访问的数据存储器地址mem_addr_o，其值就是执行阶段计算出来的地址mem_addr_i，但最后两位要设置为0，因为swl指令最多可能需要向数据存储器写入一个字，所以这里将地址对齐。

(4) 依据mem_addr_i的最后两位，确定最终要写入数据存储器的数据是读出的寄存器值reg2_i的那一部分，从而给出mem_sel_o的值，这一确定过程可以参考图9-14。

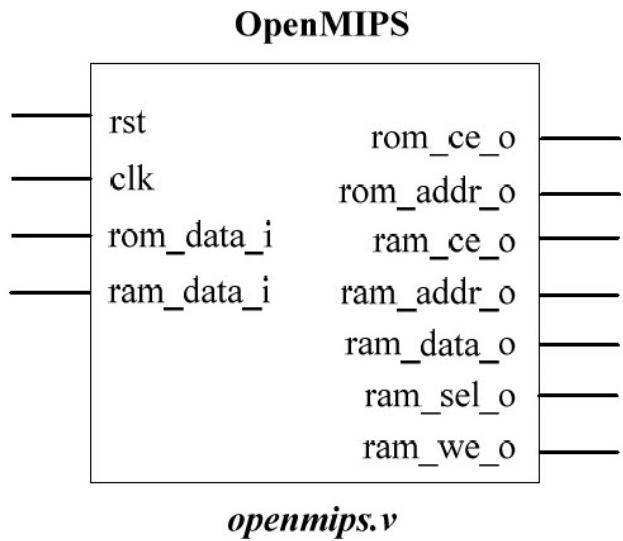
swr指令的访存过程与swl指令类似，可以对照理解。

9.3.4 修改OpenMIPS顶层模块

由于部分模块增加了接口，所以需要修改顶层模块OpenMIPS，以便将新的接口连接起来。同时，从图9-19可知，MEM模块增加了几个对数据存储器的接口，而这几个接口连接到OpenMIPS外部，所以OpenMIPS模块的接口也要做修改，新增接口如表9-6所示，修改后的OpenMIPS处理器接口图如图9-23所示，大家可以与图4-6做一对比。

表9-6 OpenMIPS模块新增加接口的描述

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	ram_data_i	32	输入	从数据存储器读取的数据
2	ram_addr_o	32	输出	要访问的数据存储器地址
3	ram_we_o	1	输出	是否是对数据存储器的写操作，为1表示是写操作
4	ram_sel_o	4	输出	字节选择信号
5	ram_data_o	32	输出	要写入数据存储器的数据
6	ram_ce_o	1	输出	数据存储器使能信号



openmips.v

图9-23 修改后的OpenMIPS处理器接口图

要修改OpenMIPS模块，将表9-6中的数据存储器接口与MEM模块的对应接口连接在一起，主要修改如下。完整代码请参考本书附带光盘中Code\Chapter9_1目录下的openmips.v文件。

```

module openmips(
    input wire                  clk,
    input  wire                  rst,
    input wire[`RegBus]          rom_data_i,
    output wire[`RegBus]         rom_addr_o,
    // 新增接口，连接数据存储器RAM
    input wire[`RegBus]          ram_data_i,
    output wire[`RegBus]         ram_addr_o,
    output wire[`RegBus]         ram_data_o,
    output wire                  ram_we_o,
)

```

```
    output wire[3:0]          ram_sel_o,
    output wire              ram_ce_o
);

.....
// 主要修改MEM模块的例化语句,
// 目的是将表9-6中的数据存储器接口与MEM模块的对应接口连接在一起
mem mem0(
    .....
// 来自数据存储器的信息
.mem_data_i(ram_data_i),
.....
// 送到数据存储器的信息
.mem_addr_o(ram_addr_o),
.mem_we_o(ram_we_o),
.mem_sel_o(ram_sel_o),
.mem_data_o(ram_data_o),
.mem_ce_o(ram_ce_o)
);
.....
```

```
endmodule
```

9.4 修改最小SOPC

为了验证上一节添加的加载存储指令是否实现正确，需要修改在第4章中设计的最小SOPC，为其添加数据存储器RAM。

9.4.1 添加数据存储器RAM

数据存储器RAM的接口如图9-24所示，还是采用左边是输入接口，右边是输出接口的方式绘制，这样便于理解。接口的含义如表9-7所示。

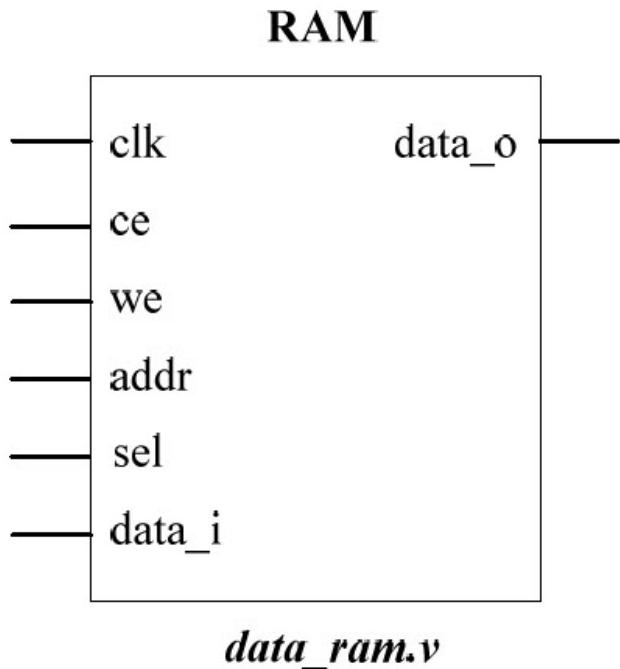


图9-24 数据存储器RAM模块接口图

表9-7 数据存储器RAM的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	ce	1	输入	数据存储器使能信号
2	clk	1	输入	时钟信号
3	data_i	32	输入	要写入的数据
4	addr	32	输入	要访问的地址
5	we	1	输入	是否是写操作, 为 1 表示是写操作
6	sel	4	输入	字节选择信号
7	data_o	32	输出	读出的数据

数据存储器RAM模块的代码如下，源文件是本书附带光盘中Code\Chapter9_1目录下的data_ram.v文件。

```
module data_ram(  
  
    input wire          clk,  
    input wire          ce,  
    input wire          we,  
    input wire[`DataAddrBus]    addr,  
    input wire[3:0]           sel,  
    input wire[`DataBus]        data_i,  
    output reg[`DataBus]       data_o  
  
);  
  
    // 定义四个字节数组  
    reg[`ByteWidth]  data_mem0[0:`DataMemNum-1];  
    reg[`ByteWidth]  data_mem1[0:`DataMemNum-1];  
    reg[`ByteWidth]  data_mem2[0:`DataMemNum-1];  
    reg[`ByteWidth]  data_mem3[0:`DataMemNum-1];
```

```
// 写操作

always @ (posedge clk) begin
    if (ce == `ChipDisable) begin
        //data_o <= ZeroWord;
    end else if(we == `WriteEnable) begin
        if (sel[3] == 1'b1) begin
            data_mem3[addr[`DataMemNumLog2+1:2]] <=
data_i[31:24];
        end
        if (sel[2] == 1'b1) begin
            data_mem2[addr[`DataMemNumLog2+1:2]] <=
data_i[23:16];
        end
        if (sel[1] == 1'b1) begin
            data_mem1[addr[`DataMemNumLog2+1:2]] <=
data_i[15:8];
        end
        if (sel[0] == 1'b1) begin
            data_mem0[addr[`DataMemNumLog2+1:2]] <=
data_i[7:0];
        end
    end
end

// 读操作

always @ (*) begin
    if (ce == `ChipDisable) begin
```

```

        data_o <= `ZeroWord;

    end else if(we == `WriteDisable) begin
        data_o <= {data_mem3[addr[`DataMemNumLog2+1:2]],
                    data_mem2[addr[`DataMemNumLog2+1:2]],
                    data_mem1[addr[`DataMemNumLog2+1:2]],
                    data_mem0[addr[`DataMemNumLog2+1:2]]};

    end else begin
        data_o <= `ZeroWord;
    end
end

endmodule

```

其中涉及到的相关宏定义在defines.v中定义，如下：

```

`define DataAddrBus      31:0          //地址总线宽度
`define DataBus          31:0          //数据总线宽度
`define DataMemNum       131071        //RAM的大小，单位是字，此处是
128K word
`define DataMemNumLog2  17             //实际使用的地址宽度
`define ByteWidth        7:0           //一个字节的宽度，是8bit

```

为了方便实现对数据存储器按字节寻址，在设计的时候使用4个8位存储器代替一个32位存储器，如图9-25所示，读操作时，从4个8位存储器中各读出一个字节，组合为一个32位的数据输出，写操作时，依据sel的值，修改其中特定存储器对应的字节即可。因此，地址addr的最低两位不需要使用，比如：读取地址n处的字，实际就是从4个8位存储器

的地址 $n/4$ 处各读取一个字节，组合起来就来地址 n 处的字。读者可以结合本节实现的数据存储器理解9.3.3节中MEM模块的输出。

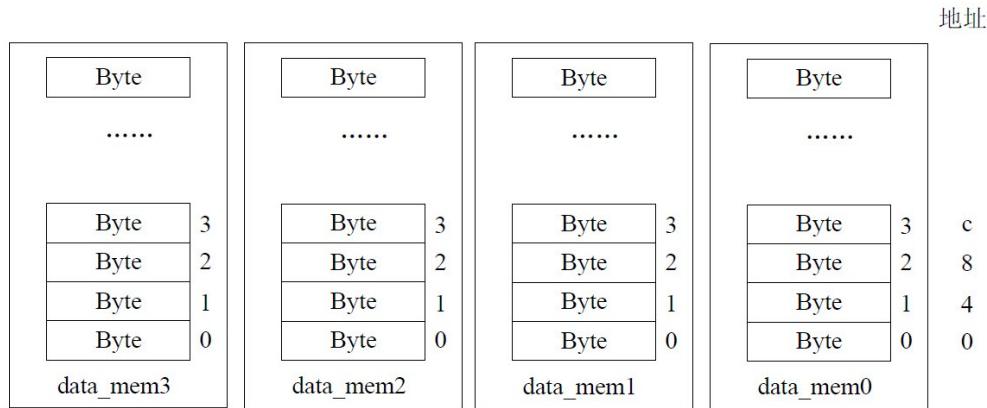


图9-25 32位数据存储器由4个8位数据存储器构成

9.4.2 修改最小SOPC

添加数据存储器RAM后的SOPC如图9-26所示。读者可以与图4-9对比。

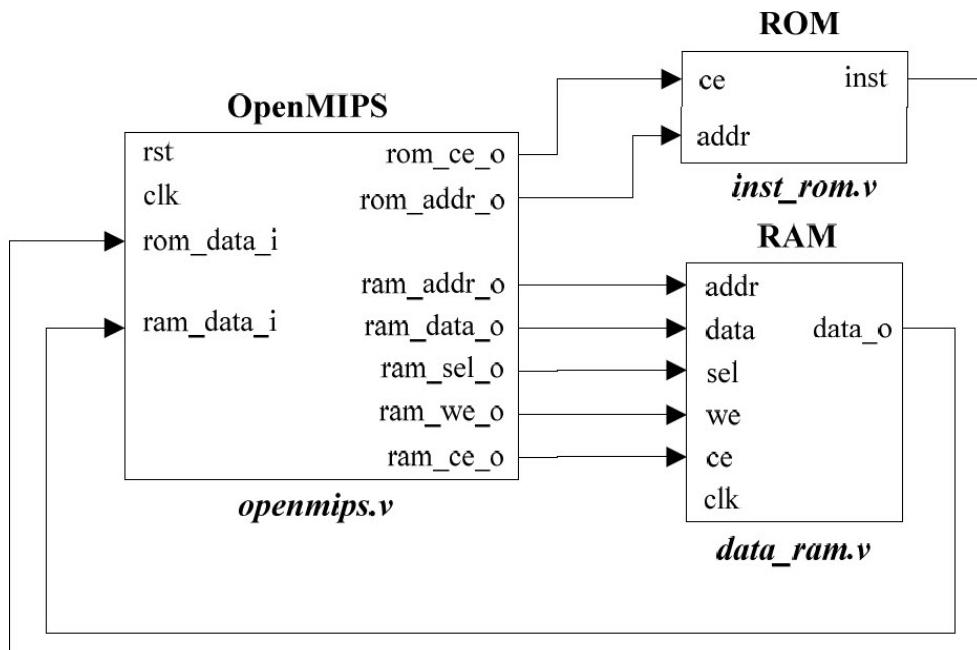


图9-26 添加数据存储器RAM后的最小SOPC

此处需要修改openmips_min_sopc.v，在其中将OpenMIPS、ROM、RAM按照图9-26所示连接起来，具体代码没有在书中列出，读者可以参考本附带光盘中Code\Chapter9_1目录下的同名文件。

9.5 测试程序

下面的测试程序是用来验证前几节实现的加载存储指令（除ll、sc指令）是否正确，程序的注释给出了预期执行效果。源文件是本书附带光盘Code\Chapter9_1\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:

#####
第一段：测试 sb 、 lb 、 lbu 指令
#####

ori  $3,$0,0xeeff      # $3 = 0x0000eeff
sb   $3,0x3($0)        # 向RAM地址0x3处存储0xff, [0x3] = 0xff

srl  $3,$3,8            # 逻辑右移8位, $3 = 0x000000ee
sb   $3,0x2($0)        # 向RAM地址0x2处存储0xee, [0x2] = 0xee
```

```

ori  $3,$0,0xccdd      # $3 = 0x0000ccdd
sb   $3,0x1($0)        # 向RAM地址0x1处存储0xdd, [0x1] = 0xdd

srl  $3,$3,8           # 逻辑右移8位, $3 = 0x0000000cc
sb   $3,0x0($0)        # 向RAM地址0x0处存储0xcc, [0x0] = 0xcc

lb    $1,0x3($0)        # 加载0x3处的字节并作符号扩展, $1 =
0xffffffff
lbu   $1,0x2($0)        # 加载0x2处的字节并作无符号扩展, $1 =
0x000000ee

#####
第二段：测试 sh 、 lh 、 lhu 指令
#####

ori  $3,$0,0xaabb      # $3 = 0x0000aabb
sh   $3,0x4($0)         # 向RAM地址0x4处存储0xaabb,
                        # [0x4] = 0xaa, [0x5] = 0xbb

lhu   $1,0x4($0)        # 加载0x4处的半字并作无符号扩展, $1 =
0x0000aabb
lh    $1,0x4($0)        # 加载0x4处的半字并作符号扩展, $1 =
0xffffaabb

ori  $3,$0,0x8899      # $3 = 0x00008899
sh   $3,0x6($0)         # 向RAM地址0x6处存储0x8899,
                        # [0x6] = 0x88, [0x7] = 0x99

```

```
lh      $1, 0x6($0)          # 加载0x6处的半字并作符号扩展，$1 =  
0xfffff8899
```

```
lhu     $1, 0x6($0)          # 加载0x6处的半字并作无符号扩展，$1 =  
0x00008899
```

```
#####第三段：测试sw、lw、lw1、lwr指令#####
```

经过上面指令的执行，此时RAM的内容如下

[0x0] = 0xcc, [0x1] = 0xdd

[0x2] = 0xee, [0x3] = 0xff

[0x4] = 0xaa, [0x5] = 0xbb

[0x6] = 0x88, [0x7] = 0x99

```
ori    $3,$0,0x4455
```

```
sll    $3,$3,0x10
```

```
ori    $3,$3,0x6677      # $3 = 0x44556677
```

```
sw    $3,0x8($0)        # 向RAM地址0x8处存储0x44556677,  
# [0x8] = 0x44, [0x9] = 0x55,  
# [0xa] = 0x66, [0xb] = 0x77
```

```
lw    $1,0x8($0)        # 加载0x8处的字，$1 = 0x44556677
```

```
lw1   $1,0x5($0)        # 非对齐加载指令lw1，执行后使得$1 =  
0xbb889977,
```

读者可以结合图9-8理解

```
lwr $1,0x8($0)          # 非对齐加载指令lwr，执行后使得$1 =  
0xbb889944,
```

读者可以结合图9-10理解

```
nop
```

```
##### 第四段：测试swl、swr指令  
#####
```

```
swr $1,0x2($0)          # 非对齐存储指令swr，执行效果如下
```

[0x0] = 0x88, [0x1] = 0x99,

[0x2] = 0x44, [0x3] = 0xff

读者可以结合图9-16理解

```
swl $1,0x7($0)          # 非对齐存储指令swl，执行效果如下
```

[0x4] = 0xaa, [0x5] = 0xbb,

[0x6] = 0x88, [0x7] = 0xbb

读者可以结合图9-14理解

```
lw $1,0x0($0)           # 加载RAM地址0x0处的字，$1 = 0x889944ff,
```

验证swr指令的执行效果

```
lw $1,0x4($0)           # 加载RAM地址0x4处的字，$1 = 0xaabb8844,
```

验证swl指令的执行效果

```
loop:
```

```
j _loop  
nop
```

上面的测试代码可分为四段，分别测试了不同的加载、存储指令，在ModelSim中的仿真效果如图9-27所示，通过观察寄存器\$1的变化，可知OpenMIPS处理器正确实现了加载存储指令。

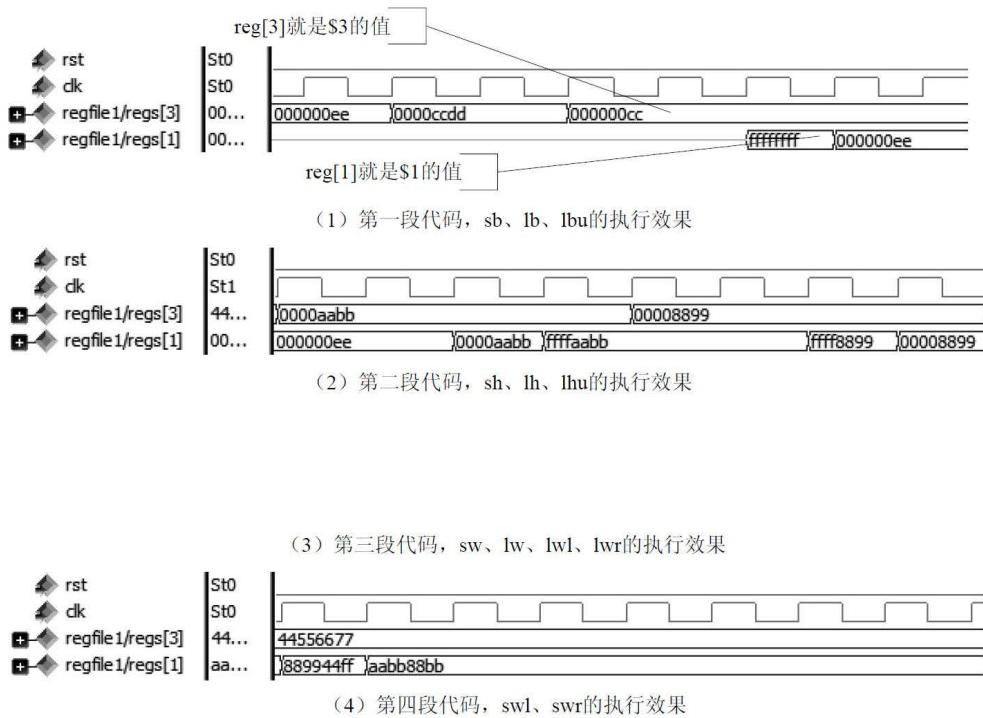


图9-27 ModelSim仿真测试加载、存储指令的执行效果

9.6 链接加载指令ll、条件存储指令sc说明

在本章前面的部分，笔者花费很多笔墨介绍了OpenMIPS中除ll、sc之外的加载、存储指令的实现过程，本节至9.9节将专门介绍链接加载

指令ll、条件存储指令sc的实现过程。ll、sc指令是MIPS32指令集架构中比较特殊的加载存储指令，用来实现信号量机制。

在多线程系统中，需要RMW（Read-Modify-Write）操作序列保证对某个资源的独占性，RMW操作序列的含义是，读取内存某个地址的数据，读取的数据经过修改，然后再保存回内存原地址，这个过程不能有任何打扰，因此需要建立一个临界区域（Critical Region），临界区域中完成的操作通常称为原子操作，原子操作不被打扰。操作系统建立临界区域的方式通常是信号量机制，如下。

wait (semaphore) ;

原子操作;

signal (semaphore) ;

semaphore是一个信号量，为1表示信号量使用中，为0表示信号量空闲。进行原子操作前，使用wait函数查询semaphore的值，如果为1，则等待，否则，将其置为1，开始执行原子操作。操作结束后，signal函数将semaphore置为0，这样其他线程就可以执行原子操作了。

需要注意的是，wait函数的执行也是一个原子操作，是一种“先检测后设置”操作（test-and-set operation），这种操作一般不希望被外部设备中断，也不希望被其他线程打断，很多处理器都有专门的指令用来实现“先检测后设置”操作，比如：680x0 CPU、x86 CPU等。这也是一种信号量机制。

MIPS32架构采用特殊的方式实现信号量机制，对于原子操作，MIPS32架构并不保证它一定是原子性的，也就是允许检测和设置在没有原子性保证的情况下运行，但只在它确实是原子的运行了的时候才让

“设置”生效。MIPS32架构采用链接加载指令ll、条件存储指令sc来实现这种信号量机制。

ll指令同一般的加载指令一样，从内存中加载一个字，但是，有一点不同，ll指令还会将处理器内部的一个链接状态位LLbit置为1，表明发生了一个链接加载操作，并将链接加载的地址保存到一个特殊寄存器LLAddr中（这个寄存器在多处理器中有作用，OpenMIPS是单处理器，所以在OpenMIPS实现过程中并没有实现LLAddr寄存器）。

ll指令执行完毕后，会进行一定的操作（如：修改加载得到的数据），然后执行sc指令，这可以认为是一个RMW序列。有如下两种情况干扰这个RMW序列，受到干扰后，处理器会设置链接状态位LLbit为0。

- 在ll、sc指令之间产生异常，从而进入异常处理例程，或者发生线程切换，导致RMW序列受到干扰。
- 多处理器的系统中，另一个CPU改写了RMW序列要操作的内存空间。对于OpenMIPS而言，只有第1种情况。

执行sc指令时，会对从ll指令开始的RMW序列进行检查，判断是否受到干扰，实际就是判断LLbit是否为1，如果没有受到任何干扰，LLbit保持为1，那么操作是原子的，sc指令会对ll指令加载数据的地址进行写回操作，并设置一个通用寄存器的值为1，表示成功，反之不进行写回操作，并设置一个通用寄存器的值为0，表示失败。

ll、sc指令的格式如图9-28所示。从图中可知，可以依据指令码对这2条指令进行区分。

31	26 25	21 20	16 15	0	
LL 110000	base	rt	offset		ll指令
SC 111000	base	rt	offset		sc指令

图9-28 ll、sc指令格式

- 当指令中的指令码为6'b110000时，是ll指令，链接加载指令。

指令用法为：ll rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字节，然后符号扩展至32位，保存到地址为rt的通用寄存器中。其中加载地址的计算方法如下。

$$\text{加载地址} = \text{signed_extended(offset)} + \text{GPR[base]}$$

此外，还要设置链接状态位LLbit为1。

- 当指令中的指令码为6'b111000时，是sc指令，条件存储指令。

指令用法为：sc rt, offset(base)。

指令作用为：如果RMW序列没有受到干扰，也就是LLbit为1，那么将地址为rt的通用寄存器的值保存到内存中指定的存储地址处，同时设置地址为rt的通用寄存器的值为1，设置LLbit为0。如果RMW序列受到了干扰，也就是LLbit为0，那么不修改内存，同时设置地址为rt的通用寄存器的值为0。其中存储地址的计算方法如下。

$$\text{存储地址} = \text{signed_extended(offset)} + \text{GPR[base]}$$

下面通过一个例子体会ll、sc指令的作用，这个例子实现了上面介绍的wait函数，不过此处是使用ll、sc指令实现的。

```
wait:  
    ori $1, $0, sem          // sem是信号量的地址，将这个地址赋给寄存器$1  
  
TryAgain:  
    ll $2, 0($1)            // 获取信号量的值，保存到寄存器$2  
    bne $2, $0, WaitForSem // 如果信号量被占用（其值为1），那么转移到地址WaitForSem  
    // 继续等待；如果信号量空闲（其值为0），那么执行下面的指令  
  
    nop  
    ori $2, $0, 1  
    sc $2, 0($1)            // 如果没有被干扰，那么设置信号量被占用（将1保存到信号  
    // 量中），同时，设置寄存器$2为1，反之，不修改信号量，  
    // 设置寄存器$2为0  
  
    beq $2, $0, TryAgain    // 如果寄存器$2为0，表示ll、sc指令没有成功，未获取到  
    // 信号量，回到TryAgain继续尝试  
    nop  
  
    jr $31                  // 反之，表示ll、sc指令成功，获取到信号量，可以进入  
    // “临界区域”了。调用wait函数时，会将返回地址放在  
    // 寄存器$31，所以此处jr $31指令就是
```

回到调用过程，

// 进入临界区域

9.7 ll、sc指令实现思路

9.7.1 ll、sc指令的实现

这2条指令都涉及访问链接状态位LLbit，可以将LLbit当做寄存器处理，ll指令需要写该寄存器，sc指令需要读该寄存器，同时，与对通用寄存器的访问一样，对LLbit寄存器的写操作也放在回写阶段进行。

ll指令在访存阶段要读取数据存储器中指定地址的数据，还要设置对LLbit寄存器的写操作，写入的值为1，这个写操作会通过MEM/WB模块传递到回写阶段，最终实现对LLbit寄存器的写。

sc指令在访存阶段要先获得LLbit寄存器的值，如果该值为1，那么会完成存储操作，同时设置对LLbit寄存器的写操作，写入的值为0，还要设置对通用寄存器rt的写操作，写入的值为1，这些写操作都会通过MEM/WB模块传递到回写阶段，最终实现对寄存器LLbit、通用寄存器rt的写；反之，如果LLbit寄存器的值为0，那么不进行存储操作，同时设置对通用寄存器rt的写操作，写入的值为0，这个写操作会通过MEM/WB模块传递到回写阶段，最终实现对通用寄存器rt的修改。

导致寄存器LLbit为0的情况有：（1）sc指令之前没有执行ll指令；
（2）ll指令执行后、sc指令执行前，发生了异常。

9.7.2 数据流图的修改

为了实现ll、sc指令，需要对数据流图作如图9-29所示的修改，主要是在回写阶段新增了一个LLbit寄存器，其中存储的就是链接状态位，只有在回写阶段才会写LLbit寄存器。同时，要将LLbit寄存器的值传递到访存阶段，以便供指令sc进行判断。

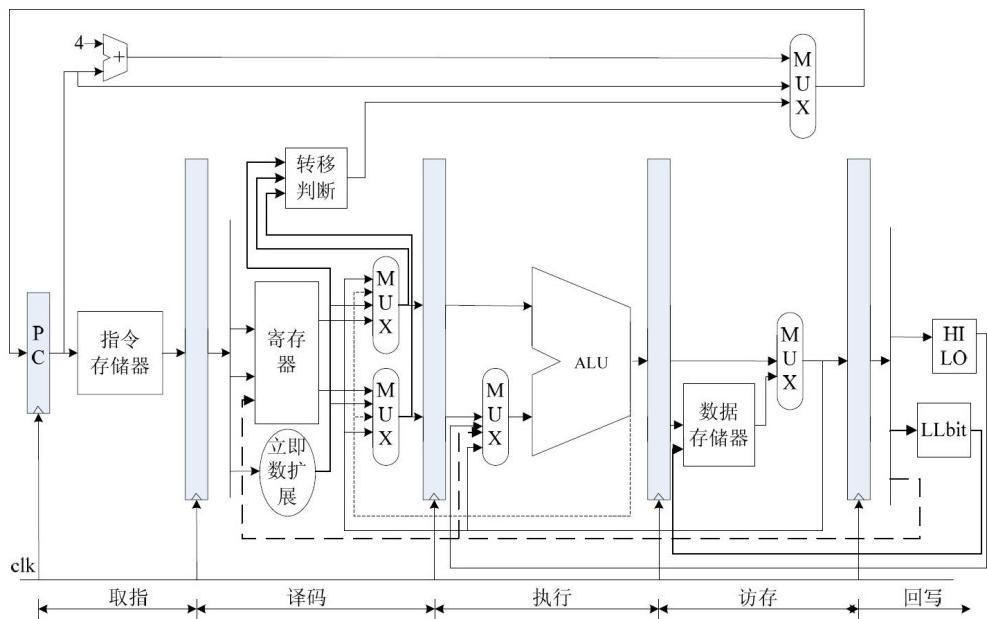


图9-29 为了实现ll、sc指令而修改的数据流图

9.7.3 系统结构的修改

为实现ll、sc指令，需要对系统结构做如图9-30所示的修改，新增了一个LLbit模块，用来实现LLbit寄存器。

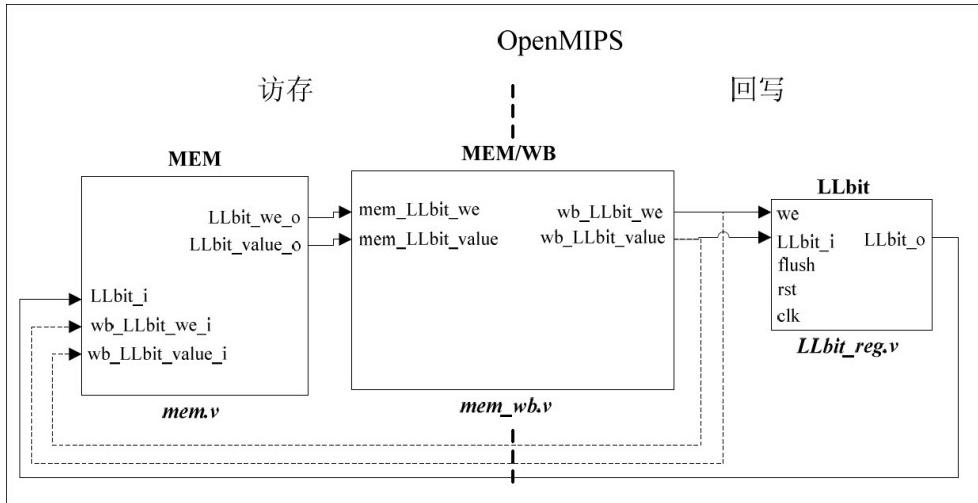


图9-30 为实现ll、sc指令而对系统结构做的修改

在访存阶段的MEM模块中会进行分析，如果是ll、sc指令，那么设置对LLbit寄存器的访问信息，通过LLbit_we_o、LLbit_value_o接口输出，前者表示是否是写操作，后者表示要写入的值，这些信息通过MEM/WB模块传递到回写阶段，最终修改LLbit寄存器。

LLbit寄存器的值通过LLbit_o接口输出到MEM模块的接口LLbit_i，当sc指令进入访存阶段时会使用该值。

需要注意的是，由于对LLbit寄存器的修改是在回写阶段最后的时钟上升沿进行的，如果直接采用LLbit模块给出的LLbit寄存器的值，可能不是正确的值，因为此时处于回写阶段的指令可能会修改LLbit寄存器，这一问题在第6章添加HI、LO寄存器时也遇到过，解决方法还是数据前推，将回写阶段指令对LLbit寄存器的操作信息前推到访存阶段，访存阶段依据这些情况，确定正确的LLbit寄存器的值，所以在图9-30中，MEM/WB模块的输出信号wb_LLbit_we、wb_LLbit_value也要送到MEM模块，就是用来解决数据相关问题的。

9.8 修改OpenMIPS以实现ll、sc指令

9.8.1 LLbit寄存器的实现

LLbit寄存器在LLbit模块中实现，模块接口如图9-30所示，各接口描述如表9-8所示。

表9-8 LLbit模块各接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	flush	1	输入	是否有异常发生
4	we	1	输入	是否要写 LLbit 寄存器
5	LLbit_i	1	输入	要写到 LLbit 寄存器的值
6	LLbit_o	1	输出	LLbit 寄存器的值

LLbit寄存器的代码如下，源文件是本书光盘中Code\Chapter9_2目录下的LLbit_reg.v文件。

```
module LLbit_reg(
    input wire      clk,
    input wire      rst,
    // 异常是否发生，为1表示异常发生，为0表示没有异常
    input wire      flush,
```

```

// 写操作

input wire      LLbit_i,
input wire      we,

// LLbit寄存器的值

output reg       LLbit_o


);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        LLbit_o <= 1'b0;
    end else if((flush == 1'b1)) begin //如果异常发生，那么设置LLbit_o为0
        LLbit_o <= 1'b0;
    end else if((we == `WriteEnable)) begin
        LLbit_o <= LLbit_i;
    end
end

endmodule

```

当有异常发生时，会使LLbit寄存器的值为0。所以此处有一个输入接口flush，当flush为1时，表示有异常发生（在第11章实现异常处理的时候将详细介绍），从而设置LLbit寄存器的值为0。

9.8.2 修改译码阶段的ID模块

在译码阶段的ID模块要增加对ll、sc指令的译码，根据图9-28给出的ll、sc指令格式可得，确定ll、sc指令的过程如图9-31所示。



图9-31 确定ll、sc指令的过程

其中涉及的宏定义如下，正是ll、sc指令的指令码，在本书附带光盘中Code\Chapter9_2目录下的defines.v文件可以找到这些定义。

```
`define EXE_LL 6'b110000  
`define EXE_SC 6'b111000
```

对译码阶段ID模块的代码做如下修改。完整代码位于本书附带光盘Code\Chapter9_2目录下的id.v文件中。

```
module id(  
    .....  
);  
    .....  
  
    always @ (*) begin  
        if (rst == `RstEnable) begin  
            .....  
        end else begin  
           aluop_o      <= `EXE_NOP_OP;
```

```

    alusel_o     <= `EXE_RES_NOP;
    wd_o          <= inst_i[15:11];           // 默认目的寄存器

地址wd_o

    wreg_o       <= `WriteDisable;
    instinvalid <= `InstInvalid;
    reg1_read_o <= 1'b0;
    reg2_read_o <= 1'b0;
    reg1_addr_o <= inst_i[25:21];           // 默认的

reg1_addr_o

    reg2_addr_o <= inst_i[20:16];           // 默认的

reg2_addr_o

    imm          <= `ZeroWord;
    . . .

case (op)

    . . .

`EXE_LL: begin                                // ll指令

    wreg_o       <= `WriteEnable;
    aluop_o      <= `EXE_LL_OP;
    alusel_o     <= `EXE_RES_LOAD_STORE;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b0;
    wd_o         <= inst_i[20:16];
    instinvalid <= `InstValid;

end

```

```

.....
`EXE_SC: begin // sc指令
    wreg_o      <= `WriteEnable;
    aluop_o     <= `EXE_SC_OP;
    alusel_o    <= `EXE_RES_LOAD_STORE;
    reg1_read_o <= 1'b1;
    reg2_read_o <= 1'b1;
    wd_o        <= inst_i[20:16];
    instvalid   <= `InstValid;
    alusel_o    <= `EXE_RES_LOAD_STORE;
end
.....
endmodule

```

译码工作主要是确定要写的目的寄存器、要读取的寄存器情况和要执行的运算等三个方面。分别介绍如下。

(1) ll指令

- 要写的目的寄存器：链接加载指令ll需要将加载结果写入通用寄存器，所以设置wreg_o为WriteEnable，同时参考图9-28可知，要写的目的寄存器是指令中的第16~20bit，所以设置wd_o为inst_i[20:16]。

- 要读取的寄存器情况：参考图9-28可知，计算加载目标地址需要使用地址为base的寄存器值，所以设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器，默认读取的寄存器地址reg1_addr_o是指令的第21~25bit，正是ll指令中的base。所以最终译码阶段的输出reg1_o就是地址为base的寄存器的值。
- 要执行的运算：设置alusel_o为EXE_RES_LOAD_STORE，表示运算类型是加载存储，设置aluop_o为EXE_LL_OP，表示运算子类型是ll。

(2) sc指令

- 要写的目的寄存器：条件存储指令sc也需要写通用寄存器，所以也设置wreg_o为WriteEnable。这一点是与其余的存储指令sb、sh、sw等的重要区别。要写的目的寄存器是指令中的第16~20bit，所以设置wd_o为inst_i[20:16]。
- 要读取的寄存器情况：参考图9-28可知，计算存储目标地址需要使用地址为base的寄存器值，所以设置reg1_read_o为1，表示通过Regfile模块的读端口1读取寄存器，默认读取的寄存器地址reg1_addr_o是指令的第21~25bit，正是sc指令中的base。所以最终译码阶段的输出reg1_o就是地址为base的寄存器的值。另外，要存储的值是地址为rt的通用寄存器的值，所以设置reg2_read_o为1，表示通过Regfile模块的读端口2读取寄存器，默认读取的寄存器地址reg2_addr_o是指令的第16~20bit，正是sc指令中的rt。所以最终译码阶段的输出reg2_o就是地址为rt的寄存器的值。
- 要执行的运算：设置alusel_o为EXE_RES_LOAD_STORE，表示运算类型是加载存储，设置aluop_o为EXE_SC_OP，表示运

算子类型是sc。

9.8.3 修改访存阶段

1. 修改MEM模块

参考图9-30可知，访存阶段的MEM模块要新增部分接口，新增接口的描述如表9-9所示。

表9-9 MEM模块新增接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	LLbit_i	1	输入	LLbit 模块给出的 LLbit 寄存器的值
2	wb_LLbit_we_i	1	输入	回写阶段的指令是否要写 LLbit 寄存器
3	wb_LLbit_value_i	1	输入	回写阶段要写入 LLbit 寄存器的值
4	LLbit_we_o	1	输出	访存阶段的指令是否要写 LLbit 寄存器
5	LLbit_value_o	1	输出	访存阶段的指令要写入 LLbit 寄存器的值

MEM模块主要修改的代码如下，完整代码请参考本书附带光盘Code\Chapter9_2目录下的mem.v文件。

```
module mem(  
    . . . . .  
    // 新增的输入接口  
    input wire          LLbit_i,  
    input wire          wb_LLbit_we_i,  
    input wire          wb_LLbit_value_i,
```

```
....  
  
// 新增的输出接口  
output reg LLbit_we_o,  
output reg LLbit_value_o,  
  
....  
  
);  
  
reg LLbit; // 保存LLbit寄存器的最新值  
....  
  
// 获取LLbit寄存器的最新值，如果回写阶段的指令要写LLbit，那么回写阶段要  
写入的  
// 值就是LLbit寄存器的最新值，反之，LLbit模块给出的值LLbit_i是最新值  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        LLbit <= 1'b0;  
    end else begin  
        if(wb_LLbit_we_i == 1'b1) begin  
            LLbit <= wb_LLbit_value_i; // 回写阶段的指令要写LLbit  
        end else begin  
            LLbit <= LLbit_i;  
        end  
    end  
end  
end
```

```

always @ (*) begin
    if(rst == `RstEnable) begin
        .....
        LLbit_we_o     <= 1'b0;
        LLbit_value_o <= 1'b0;
    end else begin
        .....
        LLbit_we_o     <= 1'b0;
        LLbit_value_o <= 1'b0;
        mem_ce_o <= `ChipDisable;
        mem_we <= `WriteDisable;
        case (aluop_i)
            .....
            `EXE_LL_OP: begin // ll指令的访存输出
                mem_addr_o     <= mem_addr_i;
                mem_we         <= `WriteDisable;
                wdata_o        <= mem_data_i;
                LLbit_we_o     <= 1'b1;
                LLbit_value_o <= 1'b1;
                mem_sel_o      <= 4'b1111;
                mem_ce_o       <= `ChipEnable;
            end
            .....
            `EXE_SC_OP: begin // sc指令的访存输出

```

```

if(LLbit == 1'b1) begin
    LLbit_we_o      <= 1'b1;
    LLbit_value_o  <= 1'b0;
    mem_addr_o     <= mem_addr_i;
    mem_we         <= `WriteEnable;
    mem_data_o    <= reg2_i;
    wdata_o        <= 32'b1;
    mem_sel_o     <= 4'b1111;
    mem_ce_o       <= `ChipEnable;
end else begin
    wdata_o        <= 32'b0;
end
end
.....
endmodule

```

MEM模块的代码增加了一个过程，以获得LLbit寄存器的最新值，然后针对ll、sc指令分别给出了对数据存储器的访问信息。

(1) ll指令

- 给出要访问的数据存储器地址mem_addr_o，其值就是执行阶段计算出来的地址mem_addr_i，参考9.3.2节。
- 因为是加载操作，所以设置mem_we_o为WriteDisable。
- 因为要访问数据存储器，所以设置mem_ce_o为ChipEnable。

- 因为是加载一个字，所以设置mem_sel_o为4'b1111。
- 要写入通用寄存器rt的值就是从数据存储器加载到的数据mem_data_i，所以设置wdata_o为mem_data_i。
- 要置LLbit寄存器为1，所以设置LLbit_we_o为1，表示要写LLbit寄存器，同时，设置LLbit_value_o为1，表示要写入LLbit寄存器的值为1。

(2) sc指令

如果LLbit寄存器的值为1，表示之前已执行过ll指令，并且在ll指令执行后、当前sc指令执行前的这段时间内，没有异常发生，此时，sc指令的访存信息如下。

- 给出要访问的数据存储器地址mem_addr_o，其值就是执行阶段计算出来的地址mem_addr_i，参考9.3.2节。
- 因为是存储操作，所以设置mem_we_o为WriteEnable。
- 因为要访问数据存储器，所以设置mem_ce_o为ChipEnable。
- 因为是存储一个字，所以设置mem_sel_o为4'b1111。
- 要存储的数据是reg2_i，是从译码阶段传递过来的，其值就是地址为rt的通用寄存器的值。
- 要置地址为rt的通用寄存器为1，所以设置wdata_o为1。
- 要置LLbit寄存器为0，所以设置LLbit_we_o为1，表示要写LLbit寄存器，同时，设置LLbit_value_o为0，表示要写入LLbit寄存器的值为0。

反之，如果LLbit的值为0，表示之前没有执行过ll指令，或者在ll指令执行后、当前sc指令执行前的这段时间内，有异常发生，此时，sc指令的访存信息如下。

- 不修改数据存储器，所以mem_we_o保持默认值WriteDisable，mem_ce_o保持默认值ChipDisable。
- 不修改LLbit寄存器的值，所以LLbit_we_o保持默认值0。
- 要置地址为rt的通用寄存器为0，所以设置wdata_o为0。

2. 修改MEM/WB模块

从图9-30可知，MEM/WB模块要新增部分接口，新增接口的描述如表9-10所示。

表9-10 MEM/WB模块新增接口的描述

序号	接口名	宽度(bit)	输入/输出	作用
1	mem_LLbit_we	1	输入	访存阶段的指令是否要写 LLbit 寄存器
2	mem_LLbit_value	1	输入	访存阶段的指令要写入 LLbit 寄存器的值
3	wb_LLbit_we	1	输出	回写阶段的指令是否要写 LLbit 寄存器
4	wb_LLbit_value	1	输出	回写阶段的指令要写入 LLbit 寄存器的值

MEM/WB模块要修改的代码如下，作用很直白：在访存阶段没有暂停时，简单地将MEM给出的对LLbit寄存器的写信息传递到访存阶段，完整代码请读者参考本书附带光盘Code\Chapter9_2目录下的mem_wb.v文件。

```
module mem_wb(
    .....
    input wire           mem_LLbit_we,
    input wire           mem_LLbit_value,
```

```
output reg                               wb_LLbit_we,
output reg                               wb_LLbit_value
);

always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        .....
        wb_LLbit_we     <= 1'b0;
        wb_LLbit_value <= 1'b0;
    end else if(stall[4] == `Stop && stall[5] == `NoStop)
begin
    .....
    wb_LLbit_we     <= 1'b0;
    wb_LLbit_value <= 1'b0;
end else if(stall[4] == `NoStop) begin // 判断访存阶段
是否暂停
    .....
    wb_LLbit_we     <= mem_LLbit_we;

```

```
    end  
end  
  
endmodule
```

9.8.4 修改OpenMIPS模块

因为一些模块增加了接口，所以要修改顶层模块OpenMIPS，以便将这些新增加的接口按照图9-30所示的关系连接起来。具体修改代码不再给出，读者可以参考本书附带光盘Code\Chapter9_2目录下的openmips.v文件。

注意一点，因为目前还没有实现异常处理，所以可以直接设置LLbit模块的输入接口flush为0，表示没有异常发生，当后续章节实现异常处理后，再将其连接到正确的模块。

9.9 测试ll、sc指令实现效果

通过如下程序测试ll、sc指令的实现效果。源文件是位于本书附带光盘中Code\Chapter9_2\AsmTest目录下的inst_rom.S文件中。

```
.org 0x0  
.set noat  
.set noreorder
```

```
.set nomacro  
.global _start  
  
_start:
```

```
##### 第一段：在没有执行lw指令的情况下，执行sc指令  
#####
```

```
ori $1,$0,0x1234    # $1 = 0x00001234  
sw  $1,0x0($0)      # 向数据存储器地址0处存储0x00001234,  
# [0x0] = 0x00001234
```

```
ori $1,$0,0x5678    # $1 = 0x00005678  
sc  $1,0x0($0)      # 因为之前没有执行lw指令，所以此处的sc指令不会  
修改数据
```

```
# 存储器。存储失败，通用寄存器$1变为0，即$1 = 0x0
```

```
lw   $1,0x0($0)      # 从数据存储器0x0处加载字，用来验证上一条sc指令  
确实没有
```

```
# 修改数据存储器，加载后使得寄存器$1 = 0x00001234  
nop
```

```
##### 第二段：模仿Read-Modify-Write过程  
#####
```

```
ori $1,$0,0x0        # $1 = 0x0  
lw  $1,0x0($0)       # 从数据存储器0x0处加载字，保存到寄存器$1,  
# 执行完毕后，使得寄存器$1 = 0x00001234
```

```
nop  
addi $1,$1,0x1      # 将读出的数据加1, $1 = 0x00001235  
sc  $1,0x0($0)      # 将修改后的数据再保存回数据存储器, 保存成功会设置寄存器$1,  
                      # 使得$1 = 0x1  
  
lw  $1,0x0($0)      # 从数据存储器0x0处加载字, 以验证是否是sc指令存储的数据,  
                      # 执行完毕后, 使得寄存器$1 = 0x00001235  
  
_loop:  
    j _loop  
    nop
```

测试代码可以分两段理解。

第一段：在没有执行ll指令的情况下，执行sc指令，此时sc指令不会修改数据存储器，并且会设置指令中的通用寄存器rt为0。

第二段：模仿Read-Modify-Write过程，首先执行ll指令以读取数据存储器地址0x0处的字，将读出的数据加1，然后通过sc指令保存回数据存储器的地址0x0处。

程序的注释已经给出了执行效果，通过观察通用寄存器\$1的变化可以验证ll、sc指令是否正确实现。ModelSim仿真如图9-32所示，从仿真结果可知，OpenMIPS处理器正确实现了ll、sc指令。

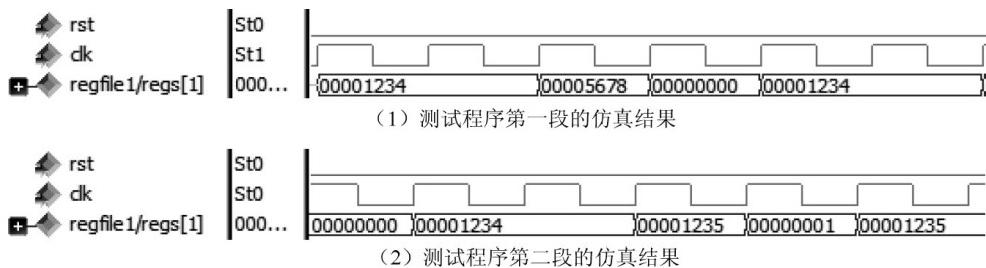


图9-32 测试程序的仿真结果

9.10 load相关问题

9.10.1 load相关问题介绍

在之前编写测试程序的时候，都很小心地避开了一个问题，那就是load相关问题，我们观察下面这段程序。

```
.....
lw    $1, 0x0($0)      // 从数据存储器的地址0x0处加载字，保存到通用寄存器
$1
beq   $1, $2, Label    // 比较通用寄存器$1与$2，如果相等，那么转移到
Label处
....
```

加载指令lw会在访存阶段从数据存储器读取数据，也就是在访存阶段才能获得要写入通用寄存器\$1的值，这个值是\$1的最新值，此时紧接着的转移指令beq处于执行阶段，而beq在上一周期译码阶段时，就已经对寄存器\$1与\$2的值进行了比较，并判断是否转移，显然这个判断依据的寄存器\$1的值不是lw指令加载得到的值，所以程序并没有按照意图运行。如图9-33所示。

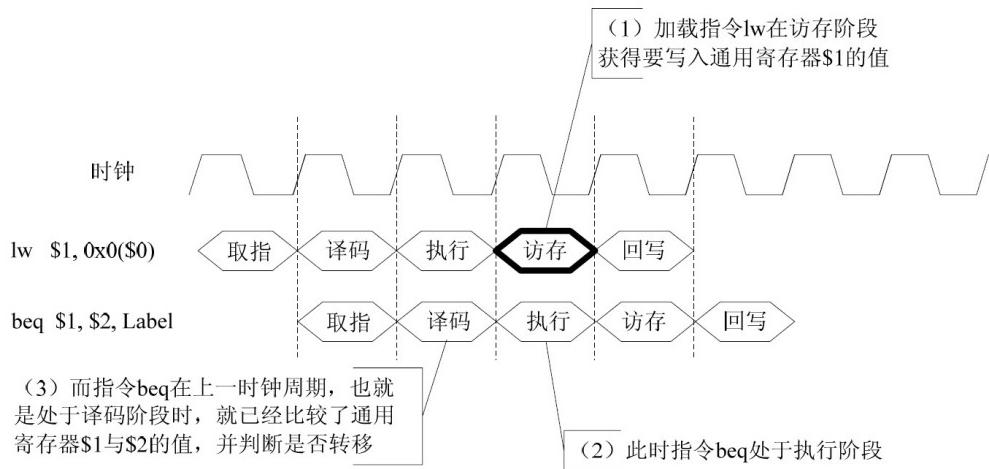


图9-33 load相关导致程序执行出错

即使通过数据前推的方法，将访存阶段加载得到的数据前推，也解决不了问题，因为数据加载时，beq指令已经处于执行阶段了，已经进行了比较判断，这种情况称为load相关。

9.10.2 解决方法

OpenMIPS解决load相关的方法是：在译码阶段检查当前指令与上一条指令是否存在load相关，如果存在load相关，那么就让流水线的译码、取指阶段暂停，而执行、访存、回写阶段继续，相当于插入一个空指令，这样处于执行阶段的加载指令会继续运行，不受影响，当其运行到访存阶段时，将加载得到的数据前推到译码阶段，然后，流水线可以继续运行。按照这个方法执行上面的示例程序，流水线会有如图9-34所示的情形。

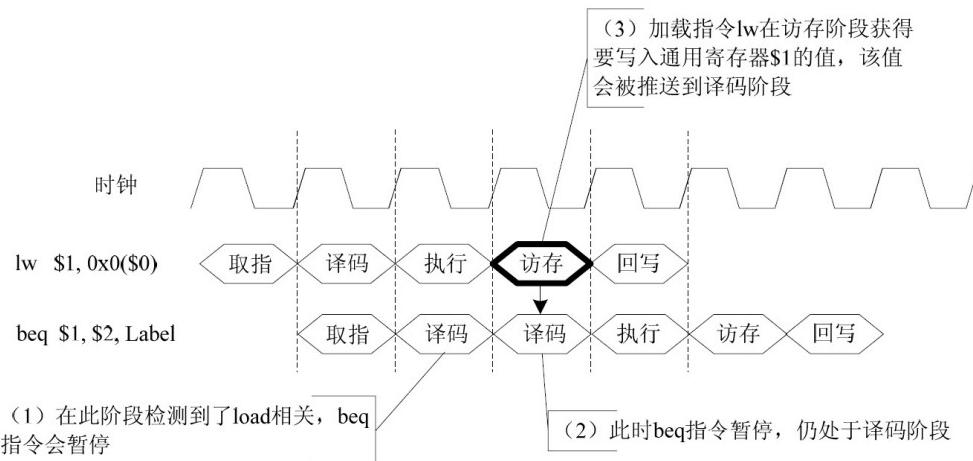


图9-34 OpenMIPS对load相关的解决方法

其中，将访存阶段的数据前推到译码阶段，这一点我们早已经实现，所以需要增加的就是判断load相关，并在出现load相关时，暂停流水线。为了实现这一新增功能，需要对OpenMIPS系统结构做如图9-35所示的修改。

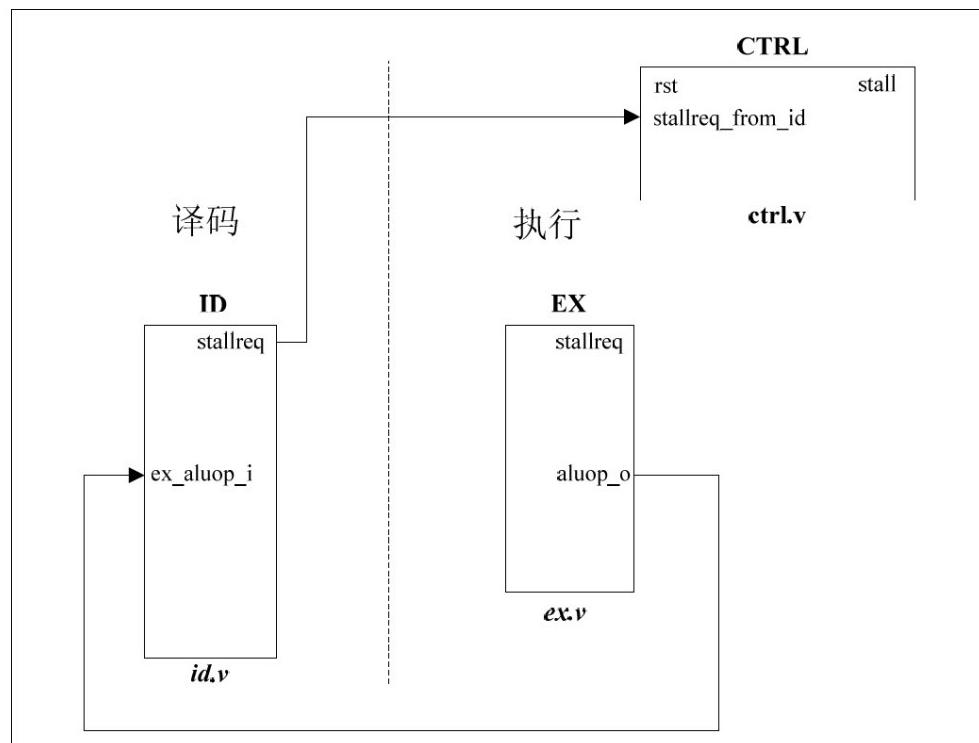


图9-35 修改OpenMIPS系统结构以解决load相关问题

将处于执行阶段的指令的运算子类型aluop_o、要写的目的寄存器地址wd_o（未在图9-35中画出，是因为在图5-8中已经画出，此处只画新增的接口）等信息传递到译码阶段的ID模块，后者据此判断是否存在load相关，如果存在load相关，那么通过stallreq接口通知CTRL模块请求流水线暂停。其中，ID模块的stallreq接口在第7章中就已引入，只是一直没有使用（参考7.5.2节）。

9.11 修改OpenMIPS以解决load相关问题

9.11.1 修改译码阶段的ID模块

参考图9-35可知，ID模块需要新增一个接口ex_aluop_i，该接口的描述如表9-11所示。

表9-11 ID模块增加的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	ex_aluop_i	8	输入	处于执行阶段指令的运算子类型

ID模块主要修改如下，完整代码请读者参考本书附带光盘中Code\Chapter9_3目录下的id.v文件。

```
module id(
```

```
.....
```



```

||

(exaluop_i == `EXE_LL_OP)

||

(exaluop_i == `EXE_SC_OP)) ?

1'b1 : 1'b0;

// 如果上一条指令是加载指令，且该加载指令要加载到的目的寄存器就是当前指令
// 要通过Regfile模块读端口1读取的通用寄存器，那么表示存在load相关，
// 设置stallreq_for_reg1_loadrelate为Stop
always @ (*) begin

    stallreq_for_reg1_loadrelate <= `NoStop;

    if(rst == `RstEnable) begin

        reg1_o <= `ZeroWord;

        end else if(pre_inst_is_load == 1'b1 && ex_wd_i ==
reg1_addr_o

            && reg1_read_o == 1'b1 ) begin

        stallreq_for_reg1_loadrelate <= `Stop;

    end
    .....
end

// 如果上一条指令是加载指令，且该加载指令要加载到的目的寄存器就是当前指令
// 要通过Regfile模块读端口2读取的通用寄存器，那么表示存在load相关，
// 设置stallreq_for_reg2_loadrelate为Stop
always @ (*) begin

```

```

stallreq_for_reg2_loadrelate <= `NoStop;

if(rst == `RstEnable) begin
    reg2_o <= `ZeroWord;
end else if(pre_inst_is_load == 1'b1 && ex_wd_i ==
reg2_addr_o
    && reg2_read_o == 1'b1 ) begin
    stallreq_for_reg2_loadrelate <= `Stop;

end
.....
end

//      stallreq_for_reg1_loadrelate      为      Stop      或      者
stallreq_for_reg2_loadrelate
// 为Stop, 都表示存在load相关, 从而要求流水线暂停, 设置stallreq为Stop
assign stallreq = stallreq_for_reg1_loadrelate |
                                         stallreq_for_reg2_loadrelate;

endmodule

```

参考图9-35可知，ID模块的输出信号stallreq会送到CTRL模块的接口stallreq_from_id，其值如果为Stop，表示译码阶段请求暂停，按照在

7.5节设置的暂停机制，会使流水线的取指、译码阶段暂停，而执行、访存、回写阶段继续，与我们设想的load相关解决方法一致。

9.11.2 修改OpenMIPS模块

因为ID模块添加了接口，所以要修改顶层模块OpenMIPS，以将新增加的接口按照图9-35所示连接起来。具体修改代码不在书中给出，读者可以参考本书光盘Code\Chapter9_3目录下的openmips.v文件。

9.12 测试load相关问题解决效果

通过如下程序测试load相关问题的解决效果。源文件是本书附带光盘中Code\Chapter9_3\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:
    ori $1,$0,0x1234      # $1 = 0x00001234
    sw  $1,0x0($0)        # 向数据存储器的地址0x0处存储0x00001234
                           # [0x0] = 0x00001234

    ori $2,$0,0x1234      # 设置寄存器$2 = 0x00001234
    ori $1,$0,0x0           # 设置寄存器$1 = 0x00000000
```

```

lw $1, 0x0($0)      # 从数据存储器的地址0x0将载数据处加到寄存器$1,
                      # 指令执行完毕后, 使得$1 = 0x00001234

beq $1,$2,Label      # 比较寄存器$1与$2, 相等, 转移到Label处
nop

ori $1,$0,0x4567
nop

Label:
ori $1,$0,0x89ab      # 设置寄存器$1 = 0x000089ab
nop

_loop:
j _loop
nop

```

如果load相关得到正确解决, 那么执行beq指令会使程序发生转移, 转移到Label处, 从而不会执行ori \$1, \$0, 0x4567这一指令, 也就是通用寄存器\$1的值不会为0x4567, 而是直接为0x89ab, ModelSim仿真结果如图9-36所示, 观察寄存器\$1的变化, 可知OpenMIPS正确解决了load相关问题。

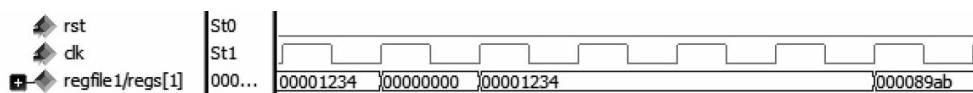


图9-36 ModelSim仿真测试结果

9.13 小结

本章的内容比较多，主要因为加载存储指令的数量比较多，而且有一些指令在其他指令集架构中很少存在类似指令，比如：ll、sc、lwl、lwr、swr、swl等，增加了理解的难度，为此，本章采用分步骤的方式实现加载存储指令，首先实现了除ll、sc指令外的一般加载存储指令。然后修改最小SOPC，为其添加了数据存储器模块，用来验证一般加载存储指令是否实现正确。接着，实现了ll、sc指令，其中引入了LLbit寄存器。最后介绍了load相关问题，并给出了OpenMIPS的解决方法。

第10章 协处理器访问指令的实现

本章首先介绍MIPS32架构中的协处理器，说明了协处理器的作用。由于OpenMIPS计划实现其中的一个协处理器——CP0，所以10.2节专题介绍CP0，然后在10.3节实现协处理器CP0，其实现方式有点类似HI、LO寄存器的实现方式。10.4节说明协处理器访问指令mfc0、mtc0的格式、作用、用法。10.5节给出了协处理器访问指令的实现思路，以及对系统结构的修改。10.6节通过修改OpenMIPS，实现了协处理器访问指令，最后编写测试程序，在ModelSim中进行仿真验证。

10.1 协处理器介绍

协处理器一词通常用来表示处理器的一个可选部件，负责处理指令集的某个扩展，具有与处理器核独立的寄存器。MIPS32架构提供了最多4个协处理器，分别是CP0~CP3，作用如表10.1所示。

表10.1 MIPS32架构定义的协处理器及其作用

协处理器	作用
CP0	系统控制
CP1	FPU
CP2	特定实现
CP3	FPU

协处理器CP0用作系统控制，CP1、CP3用作浮点处理单元，而CP2被保留用于特定实现。除CP0外的协处理器都是可选的，OpenMIPS没有实现浮点运算，所以CP1、CP3不用实现，CP2也没有作用，不用实现。而CP0是不可选的，需要实现，所以下面重点介绍协处理器CP0。

截至本章，我们的OpenMIPS处理器实现了很多指令，但这些指令都是用来运算的，实际的处理器还要支持其他广泛的操作，例如：中断处理、提供可选的配置、观察并控制系统缓存或时钟、地址转换等。MIPS32架构定义的协处理器CP0的作用就是协助实现上述的广泛操作。CP0负责的主要工作如下。

- **配置CPU工作状态：**符合MIPS32架构的硬件通常是很灵活的，可以通过读/写一个或一些内部寄存器来改变一些很根本的CPU特性（如：将字节次序从MSB变为LSB，或者从LSB变为MSB）。
- **高速缓存控制：**符合MIPS32架构的CPU一般会集成缓存控制器，用来控制、读、写缓存。
- **异常控制：**异常发生时的检测和处理都由CP0中的一些控制寄存器来定义和控制。
- **存储管理单元控制：**对系统的存储区域进行合理的控制、管理和分配，主要是对MMU、TLB的一些配置、管理、访问。
- **其他：**当要把额外的功能集成在CPU中，但又不方便当作外设访问时，常常在CP0中增加一些模块以实现这些功能。例如：时钟、时间计数器、奇偶校验错误检测等。

10.2 协处理器CP0中的寄存器

CP0中有一系列寄存器用来完成上述工作。如表10-2所示。

表10-2 CP0中的寄存器描述

标 号	寄存器助记符/名 称	功能描述	备 注
0	Index	TLB阵列的入口索引	
1	Random	产生TLB阵列的随机入口索引	
2	EntryLo0	偶数虚拟页的入口地址的低位部分	
3	EntryLo1	奇数虚拟页的入口地址的低位部分	
4	Context	指向内存虚拟页表入口地址的指针	这些都是与内存管理MMU、地址翻译快表TLB有关的寄存器
5	PageMask	控制TLB入口中可变页面的大小	
6	Wired	控制固定的TLB入口的数目	

7	保留		
8	BadVAddr	记录最近一次地址相关异常的地址	
9	<i>Count</i>	处理器计数周期	
10	EntryHi	TLB入口地址的高位部分	与内存管理MMU、地址翻译快表TLB有关的寄存器
11	<i>Compare</i>	定时中断控制	
12	<i>Status</i>	处理器状态和控制寄存器，包括决定CPU特权等级，使能哪些中断等字段	
13	<i>Cause</i>	保存上一次异常原因	
14	<i>EPC</i>	保存上一次异常时的程序计数器	
15	<i>PRId</i>	处理器标志和版本	

16	<i>Config</i>	配置寄存器，用来设置CPU的参数	
17	LLAddr	加载链接指令要加载的数据存储器地址	
18	WatchLo	观测点watchpoint地址的低位部分	
19	WatchHi	观测点watchpoint地址的高位部分	与调试有关
20-22	保留		
23	Debug	调试控制和异常状况	与调试有关
24	DEPC	上一次调试异常的程序计数器	与调试有关
25	保留		
26	ErrCtl	控制Cache指令访问数据和SPRAM	与Cache有关
27	保留		
28	TagLo/DataLo	Cache中Tag接口	与Cache有

		的低位部分	关
29	保留		
30	ErrorEPC	上一次系统错误时的程序计数器	
31	DESAVE	用于调试处理的暂停寄存器	与调试有关

从表中可以发现有很多寄存器都是与缓存、MMU、TLB、调试有关的，而OpenMIPS的设计目标是一个轻量级的处理器，并不打算实现缓存、MMU、TLB、调试等复杂功能，所以相关的寄存器都可以不用实现，本书也不再介绍这些寄存器，读者只需注意表中使用加粗、斜体标注的7个寄存器即可，下面依次介绍这7个寄存器的格式、作用。

1. Count寄存器（标号为9）

Count寄存器是一个不停计数的32位寄存器，计数频率一般与CPU时钟频率相同，当计数达到32位无符号数的上限时，会从0开始重新计数。Count寄存器可读、可写。其字段如表10-3所示。

表10-3 Count寄存器的字段

Bit	31-0
标志名	Count

2. Compare寄存器（标号为11）

Compare寄存器是一个32位的寄存器，与Count寄存器一起完成定时中断功能。当Count寄存器中的计数值与Compare寄存器中的值一样时，会产生定时中断。这个中断会一直保持，直到有数据被写入Compare寄存器。Compare寄存器可读、可写。其字段如表10-4所示。

表10-4 Compare寄存器的字段

Bit	31-0
标志名	Compare

3. Status寄存器（标号为12）

Status寄存器也是一个32位、可读、可写的寄存器，用来控制处理器的操作模式、中断使能以及诊断状态。其字段如表10-5所示。

表10-5 Status寄存器的各个字段

Bit	31-28	27	26	25	24-23	22	21	20	19
标志名	CU3-CU0	RP	R	RE	0	BEV	TS	SR	NMI
Bit	18-16	15-8		7-5	4	3	2	1	0
标志名	0	IM7-IM0		R	UM	R	ERL	EXL	IE

表10-5中标识为R的字段是保留字段，下面逐一介绍其中的非保留字段。读者朋友如果没有时间，可以只理解其中使用灰色背景的字段，OpenMIPS处理器也只实现了这些字段。

- CU3-CU0

表示协处理器是否可用（Coprocessor Usability），分别控制协处理器CP3、CP2、CP1、CP0。为0时，表示相应的协处理器不可用；为

1时，表示相应的协处理器可用。对于OpenMIPS处理器而言，只有协处理器CP0，所以可以设置本字段为4'b0001。

- RP

表示是否启用低功耗模式（Reduced Power），但是否实现以及如何实现是同具体处理器相关的，比如：有的处理器可以通过降低工作频率、工作电压，来实现低功耗，OpenMIPS处理器没有实现这些功能，所以本字段并没有作用。

- RE

用来改变用户态模式下的字节次序，1表示改变，0表示不改变，MIPS处理器可以在复位的时候配置工作在大端模式（MSB）还是小端模式（LSB），工作在用户态模式下的软件可以通过设置此处的RE字段，改变大小端模式。OpenMIPS处理器没有实现此项功能，固定工作在大端模式，所以本字段并没有作用。

- BEV

表示是否使用启动异常向量（Bootstrap Exception Vector），为0表示使用一般异常向量，为1表示使用启动异常向量，主要区别是：启动异常向量对应的异常处理例程入口地址位于不能被缓存、不能被MMU映射的内存空间。在系统刚刚启动的时候，缓存、MMU都没有准备好，此时只能使用启动异常向量，否则可能会出错。对OpenMIPS处理器而言，没有缓存、也没有MMU，采用的异常处理机制也相对简化了许多（在第11章会详述），所以本字段并没有作用。

- TS

表示是否关闭TLB（TLB Shutdown），为1表示关闭TLB，为0表示打开TLB。OpenMIPS处理器没有实现TLB，所以本字段并没有作用。

- SR

表示是否是软重启（Soft Reset），为1表示重启异常是由软重启引起的。

- NMI

表示是否是不可屏蔽中断（Non-Maskable Interrupt），为1表示重启异常是由不可屏蔽中断引起的。

- IM7-IM0

表示是否屏蔽相应中断（Interrupt Mask），0表示屏蔽，1表示不屏蔽，MIPS处理器可以有8个中断源，对应IM字段的8位，其中6个中断源是处理器外部硬件中断，另外2个是软件中断，中断是否能够被处理器响应是由Status寄存器与Cause寄存器共同决定的，如果Status寄存器的IM字段与Cause寄存器的IP字段的相应位都为1，而且Status寄存器的IE字段也为1时，处理器才响应相应中断。

- UM

表示是否为用户模式（User Mode），为1表示处理器运行在内核模式，为0表示处理器运行在用户模式。OpenMIPS处理器在实现的时候并没有区分内核模式、用户模式，两种模式下的权限是一样的，都可以访问处理器的所有资源，所以本字段并没有作用。

- ERL

表示是否处于错误级，当处理器接收到坏的数据时设置本字段为1。有一些MIPS处理器在接收来自缓存或内存中的数据块时，能够检验数据中附带的奇偶校验位或纠错码，当发现数据错误且无法纠正时，处理器就设置ERL字段为1，并进入奇偶校验\ECC错误的异常处理过程，这是一个特殊的异常处理过程（有别于一般的异常处理过程）。读者只需知道，OpenMIPS处理器没有对奇偶校验位或纠错码的检验过程，所以不用考虑ERL字段。

- EXL

表示是否处于异常级（Exception Level），当异常发生时，会设置本字段为1，表示处理器处于异常级，此时，处理器会进入内核模式下工作，并且禁止中断。

- IE

表示是否使能中断（Interrupt Enable），这是全局中断使能标志位。为1表示中断使能，为0表示中断禁止。

4. Cause寄存器（标号为13）

Cause寄存器主要记录最近一次异常发生的原因，也控制软件中断请求。Cause寄存器的各字段如表10-6所示，除了IP[1:0]、IV和WP，其余字段都是只读的。

表10-6 Cause寄存器的各个字段

Bit	31	30	29-28	27	26	25-24	23	22	21-16
标志名	BD	R	CE	DC	PCI	0	IV	WP	0
Bit	15-10			9-8		7	6-2		1-0
标志名	IP[7:2]			IP[1:0]		0	ExcCode		0

表10-6中标识为R的字段是保留字段，下面逐一介绍其中的非保留字段。读者朋友如果没有时间，可以只理解其中使用灰色背景的字段，OpenMIPS处理器也只实现了这些字段。

- BD

当发生异常的指令处于分支延迟槽（Branch DelaySlot）时，该字段被置为1。

- CE

当协处理器不可用异常发生时，将发生协处理器错误（Coprocessor Error）的协处理器序号存储到本字段。

- DC

这是在MIPS32/64架构中新增加的字段，将其置为1，可以使Count寄存器停止计数，这样做的目的之一是减少功耗。

- PCI

这是在MIPS32/64架构中新增加的字段，当协处理器CP0的性能计数器溢出时（Performance Count Interrupt），设置本字段为1，以产生中断。

- IV

中断向量（Interrupt Vector）的选择与此字段有关，将该字段置为0表示使用一般中断向量，反之，表示使用特殊中断向量。OpenMIPS处理器通过一种简单的异常向量表的方式来处理中断（在第11章会有详述），所以这个字段没有作用。

- WP

观测挂起（Watch Pending）字段，该字段与调试有关，为1表示有一个观测点被触发，处理器处于异常模式。

- IP[7:2]

中断挂起（Interrupt Pending）字段，相应位用来指明外部硬件中断是否发生，1表示发生，0表示没有发生。本字段的6位与外部硬件中断的对应关系如下。

IP[7] ——5号硬件中断

IP[6] ——4号硬件中断

IP[5] ——3号硬件中断

IP[4] ——2号硬件中断

IP[3] ——1号硬件中断

IP[2] ——0号硬件中断

- IP[1:0]

也是中断挂起字段，但是对应的是软件中断。

IP[1] ——1号软件中断

IP[0] ——0号软件中断

- ExcCode

本字段是一个5位的编码，用来记录发生了哪种异常，ExcCode编码的含义如表10-7所示。其中很多与MMU、Cache、TLB等模块有关，而我们的OpenMIPS处理器并没有实现这些模块，所以相应的异常也不用考虑。读者朋友如果没有时间，可以只理解其中使用灰色背景的ExcCode编码，OpenMIPS只实现了对这几种异常的处理，所以也只使用这些编码。

表10-7 ExcCode的编码及其含义

ExcCode编 码	助记符	描 述
0	Int	中断
1	Mod	TLB修改异常或者保留
2	TLBL	TLB加载异常或者取指异常或者保留
3	TLBS	TLB存储异常或者保留
4	AdEL	加载或取指过程中，地址错误异常
5	AdES	存储过程中，地址错误异常
6	IBS	取指过程中，总线错误异常

7	DBE	加载或存储数据过程中，总线错误异常
8	Sys	系统调用指令Syscall
9	Bp	断点异常
10	RI	执行未定义指令引起的异常
11	CpU	协处理器不可用异常
12	Ov	整数溢出异常
13	Tr	自陷指令引起的异常
14-22	-	保留
23	WATCH	访问WatchHi\WatchLo地址
24	MCheck	机器检测，CPU检测到CPU控制系统中的灾难性错误
25-31	-	保留

5. EPC寄存器（标号为14）

EPC是异常程序计数器（Exception Program Counter），用来存储异常返回地址，一般情况下，存储发生异常的指令的地址，但是，如果发生异常的指令位于延迟槽中，那么EPC存储的是前一条转移指令的地址。该寄存器可读、可写。其字段如表10-8所示。

表10-8 EPC寄存器的字段

Bit	31-0
标志名	EPC

6. PRId寄存器（标号为15）

PRId寄存器是处理器标志（Processor Identifier）寄存器，包含的信息有：制造商信息、处理器类型以及处理器的版本等。各个字段如表10-9所示。其中R是保留字段。

表10-9 PRId寄存器的各个字段

Bit	31-24	23-16	15-6	5-0
标志名	R	Company ID	Processor ID	Revision

各个字段的含义如下。

- Company ID

指明设计或生产该处理器的公司。

- Processor ID

指明处理器的类型，软件可以依据这个字段来区分不同类型的MIPS处理器。

- Revision

指明处理器的版本号，软件可以依据这个字段来区分同类型处理器的不同版本。

7. Config寄存器（标号为16）

Config寄存器包含了与处理器有关的各种配置和功能信息，其各个字段如表10-10所示，大部分字段由硬件在重启时进行初始化，或定为常量。

表10-10 Config寄存器的各个字段

Bit	31	30-16	15	14-13	12-10	9-7	6-4	3	2-0
标志名	M	Impl	BE	AT	AR	MT	0	VI	K0

各个字段的含义如下。有一些字段是与MMU、TLB、Cache等模块有关，读者朋友如果没有时间，可以只理解其中使用灰色背景的字段。

- M

表示是否存在Config1寄存器，MIPS32架构中实际定义了4个配置寄存器：Config、Config1-3，OpenMIPS处理器只实现了Config寄存器，所以OpenMIPS在初始化的时候需要设置本字段为0，表示没有Config1寄存器。

- Impl

这是与实现相关的配置标记，MIPS32架构的处理器会在本字段设置一些自定义的信息，OpenMIPS处理器没有使用本字段。

- BE

其值为1表示处理器工作在大端模式（MSB），为0表示处理器工作在小端模式（LSB）。OpenMIPS处理器工作在大端模式，所以设置本字段为1。

- AT

指令集架构类型（Architecture Type）字段，当其值为 $2'b00$ 时，表示MIPS32架构。

- AR

指令集架构发行版本（Architecture Revision）字段，当其值为 $3'b000$ 时，表示MIPS32/64架构的发行版1，当其值为 $3'b001$ 时，表示MIPS32/64架构的发行版2。

- MT

MMU类型字段，当其值为 $3'b000$ 时，表示没有MMU。OpenMIPS处理器没有实现MMU，所以本字段固定为 $3'b000$ 。

- VI

如果一级指令缓存是使用虚拟程序地址索引做标签，那么就设置本字段为1。OpenMIPS处理器没有实现缓存，所以本字段固定为0。

- K0

表示内存的Kseg0区域是否可缓存，其中Kseg0是内存中的一段空间，读者不用了解具体细节，因为OpenMIPS处理器没有实现缓存，所以本字段固定为 $3'b000$ 。

以上就是协处理器CP0中的主要寄存器，也是实现OpenMIPS处理器的CP0中的寄存器，读者可能会有疑问，在之前的介绍中提到CP0是用来系统控制的，但是截止到现在似乎只是介绍了CP0中的寄存器，那么CP0是如何实现系统控制功能的呢？其实，CP0的控制功能就是通过上面介绍的寄存器实现的，比如：状态寄存器Status中的中断掩码字段IM，处理器要依据该字段处理发生的中断，通过修改这个字段，就可以控制哪些中断处理，哪些中断不处理，这就体现了CP0的控制功能。中断处理的详细过程，会在第11章介绍。

10.3 协处理器CP0的实现

要实现协处理器访问指令，首先要实现协处理器CP0，其实现方式类似于第6章中HI、LO寄存器的实现方式。CP0的接口如图10-1所示。这里采用左边是输入接口，右边是输出接口的方式绘制，这样比较直观，便于理解。各接口的描述如表10-11所示。

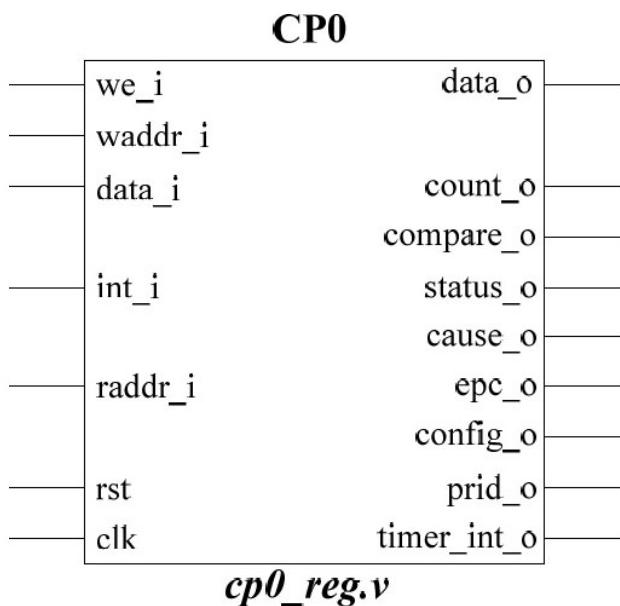


图10-1 协处理器CP0的接口图

表10-11 协处理器CP0的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	raddr_i	5	输入	要读取的 CP0 中寄存器的地址
4	int_i	6	输入	6 个外部硬件中断输入
5	we_i	1	输入	是否要写 CP0 中的寄存器
6	waddr_i	5	输入	要写的 CP0 中寄存器的地址
7	wdata_i	32	输入	要写入 CP0 中寄存器的数据
8	data_o	32	输出	读出的 CP0 中某个寄存器的值
9	count_o	32	输出	Count 寄存器的值
10	compare_o	32	输出	Compare 寄存器的值
11	status_o	32	输出	Status 寄存器的值
12	cause_o	32	输出	Cause 寄存器的值

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
13	epc_o	32	输出	EPC 寄存器的值
14	config_o	32	输出	Config 寄存器的值
15	prid_o	32	输出	PRId 寄存器的值
16	timer_int_o	1	输出	是否有定时中断发生

协处理器 CP0 的实现代码如下，源文件是本书附带光盘 Code\Chapter10 目录下的 cp0_reg.v 文件。

```
module cp0_reg(
    input wire                  clk,
    input wire                  rst,
    input wire                  we_i,
    input wire[4:0]              waddr_i,
    input wire[4:0]              raddr_i,
    input wire[`RegBus]          data_i,
```

```

    input wire[5:0]           int_i,
    output reg[`RegBus]       data_o,
    output reg[`RegBus]       count_o,
    output reg[`RegBus]      compare_o,
    output reg[`RegBus]      status_o,
    output reg[`RegBus]      cause_o,
    output reg[`RegBus]      epc_o,
    output reg[`RegBus]      config_o,
    output reg[`RegBus]      prid_o,
    output reg               timer_int_o
);

//*****
** 第一段：对CP0中寄存器的写操作 ****
***** */


```

```

always @ (posedge clk) begin
    if(rst == `RstEnable) begin

```

```

        //Count寄存器的初始值，为0
        count_o <= `ZeroWord;

```

```
//Compare寄存器的初始值，为0
compare_o <= `ZeroWord;

//Status寄存器的初始值，其中CU字段为4'b0001，表示协处理器
CP0存在
status_o <= 32'b00010000000000000000000000000000;

//Cause寄存器的初始值
cause_o <= `ZeroWord;

//EPC寄存器的初始值
epc_o <= `ZeroWord;

//Config寄存器的初始值，其中BE字段为1，表示工作在大端模式
(MSB)
config_o <= 32'b00000000000000001000000000000000;

//PRId寄存器的初始值，其中制作者是L，对应的是0x48（自行定义
的）
//类型是0x1，表示是基本类型，版本号是1.0
prid_o <= 32'b0000000010011000000000100000010;

timer_int_o <= `InterruptNotAssert;

end else begin
```

```
count_o <= count_o + 1 ;      //Count 寄存器的值在每个时钟周期加1

cause_o[15:10] <= int_i;     //Cause的第10~15bit 保存外部中断声明

//当Compare寄存器不为0，且Count寄存器的值等于Compare寄存器的值时，

//将输出信号timer_int_o置为1，表示时钟中断发生

if(compare_o != `ZeroWord && count_o == compare_o) begin
    timer_int_o <= `InterruptAssert;
end

if(we_i == `WriteEnable) begin
    case (waddr_i)
```

```
 `CP0_REG_COUNT: begin //写Count寄存器
    count_o <= data_i;
end

`CP0_REG_COMPARE: begin //写Compare寄存器
    compare_o <= data_i;

timer_int_o <= `InterruptNotAssert;

end

`CP0_REG_STATUS: begin //写Status寄存器
    status_o <= data_i;
end

`CP0_REG_EPC: begin //写EPC寄存器
    epc_o <= data_i;
end

`CP0_REG_CAUSE: begin //写Cause寄存器
    //Cause寄存器只有IP[1:0]、IV、WP字段是可写的
    cause_o[9:8] <= data_i[9:8];
    cause_o[23] <= data_i[23];
    cause_o[22] <= data_i[22];
```

```

        end
    endcase
end
end
end

//*****************************************************************************
** 第二段：对CP0中寄存器的读操作 ****
** */

always @ (*) begin
    if(rst == `RstEnable) begin
        data_o <= `ZeroWord;
    end else begin
        case (raddr_i)
            `CP0_REG_COUNT: begin          //读Count寄存
器
                data_o <= count_o ;
            end
            `CP0_REG_COMPARE: begin       //读Compare寄存器
                data_o <= compare_o ;
            end
            `CP0_REG_STATUS: begin        //读Status寄存器
                data_o <= status_o ;
            end

```

```

`CP0_REG_CAUSE: begin           //读Cause寄存器
    data_o <= cause_o ;
end

`CP0_REG_EPC: begin            //读EPC寄存器
    data_o <= epc_o ;
end

`CP0_REG_PRId: begin          //读PRId寄存器
    data_o <= prid_o ;
end

`CP0_REG_CONFIG: begin         //读Config寄存器
    data_o <= config_o ;
end

default: begin
end

endcase

end

endmodule

```

其中涉及的宏定义在defines.v中定义，如下：

```

//定义CP0中各个寄存器的地址，与表10-2中的标号是一致的
`define CP0_REG_COUNT      5'b01001
`define CP0_REG_COMPARE    5'b01011
`define CP0_REG_STATUS     5'b01100

```

```
`define CP0_REG_CAUSE      5'b01101
`define CP0_REG_EPC        5'b01110
`define CP0_REG_PRId       5'b01111
`define CP0_REG_CONFIG     5'b10000

`define InterruptAssert   1'b1
`define InterruptNotAssert 1'b0
```

上述代码可以分为两段理解，第一段实现了对CP0中寄存器的写操作，依据写入地址，将输入数据保存到不同的寄存器中，这是一个时序逻辑；第二段实现了对CP0中寄存器的读操作，依据读取地址，将相应寄存器的值通过data_o接口输出，这是一个组合逻辑。注意以下几点。

(1) OpenMIPS只实现了CP0中的Count、Compare、Status、Cause、EPC、PRId、Config 7个寄存器。

(2) 这7个寄存器中的PRId、Config不可以写，所以在第一段代码中没有写入这两个寄存器的代码。此外，Cause寄存器只有其中的IP[1:0]、IV、WP三个字段可写，所以对Cause寄存器的写入是选择性的。

(3) Count寄存器的值在每个时钟周期都会加1。

(4) 当Compare寄存器不为0，且Count寄存器的值等于Compare寄存器的值时，将输出信号timer_int_o置为1，表示时钟中断发生，这个中断会一直声明，直到有数据写入Compare寄存器。

(5) 当写Compare寄存器的时候，会将输出信号timer_int_o置为0，表示取消时钟中断的声明。

(6) MIPS32架构支持8个中断，但是有2个是软件中断，支持的外部硬件中断只有6个，所以CP0模块的中断输入信号int_i的宽度是6。

(7) Cause寄存器的第10~15bit是IP[7:2]，也就是外部硬件中断挂起字段，指明外部硬件中断是否挂起，所以代码中直接将外部中断输入int_i赋给Cause寄存器的第10~15bit。

(8) CP0中寄存器的地址与表10-2中的标号是一致的。

10.4 协处理器访问指令说明

要实现CP0的控制功能，需要对CP0中的有关寄存器进行设置，这涉及对CP0中寄存器的访问，需要使用协处理器访问指令。MIPS32指令集架构中定义了2条协处理器访问指令：mtc0、mfc0，前者实现修改CP0中的寄存器，后者实现读取CP0中的寄存器。指令格式如图10-2所示。

31	26 25	21 20	16 15	11 10	3 2	0	
COP0 010000	MT 00100	rt	rd	00000000	sel		mtc0指令
COP0 010000	MF 00000	rt	rd	00000000	sel		mfc0指令

图10-2 mtc0、mfc0指令格式

从图10-2中可以发现，这2条指令的格式与之前已实现的指令都不同，主要特点是：指令码都为6'b010000，MIPS32指令集架构定义为

COP0类，需要借助于第21~25bit的值才能确定具体是哪一条指令。此外，指令的第3~10bit为0，第0~2bit是sel域，这个域的作用取决于具体的MIPS32架构处理器，对OpenMIPS处理器而言，sel域没有作用，不用考虑。下面分别说明mtc0、mfc0两条指令的用法、作用。

- 当指令码是6'b010000，且第21~25bit的值为5'b00100时，是mtc0指令。

指令用法为：mtc0 rt, rd。

指令作用为：CPR[0, rd] <- GPR[rt]，将地址为rt的通用寄存器的值赋给协处理器CP0中地址为rd的寄存器。

- 当指令码是6'b010000，且第21~25bit的值为5'b00000时，是mfc0指令。

指令用法为：mfc0 rt, rd。

指令作用为：GPR[rt] <- CPR[0, rd]，读出协处理器CP0中地址为rd的寄存器的值，并赋给地址为rt的通用寄存器。

10.5 协处理器访问指令实现思路

10.5.1 实现思路

与对HI、LO寄存器的访问一样，对CP0中所有寄存器的写操作也都放在回写阶段。

1. mtc0实现思路

- (1) 在译码阶段依据指令，读出地址为rt的通用寄存器的值。
- (2) 在执行阶段确定要写入CP0中寄存器的值，其实就是译码阶段读出的地址为rt的通用寄存器的值，将这些信息传递到访存阶段。
- (3) 访存阶段再将这些信息传递到回写阶段。
- (4) 回写阶段依据这些信息修改CP0中的地址为rd的寄存器。

2. mfc0实现思路

- (1) 在执行阶段获取CP0中指定寄存器的值，作为要写入目的通用寄存器的数据，并将这些信息传递到访存阶段。
- (2) 访存阶段再将这些信息传递到回写阶段。
- (3) 回写阶段依据这些信息修改地址为rt的通用寄存器。

10.5.2 数据流图的修改

添加协处理器CP0后的数据流图如图10-3所示。相比图9-29，在回写阶段增加了CP0模块，并且CP0模块的输出数据传递到执行阶段，用于确定最后参与运算的操作数。比如：mfc0指令在执行阶段就会选择从CP0传递过来的数据，作为运算结果，写入目的寄存器。

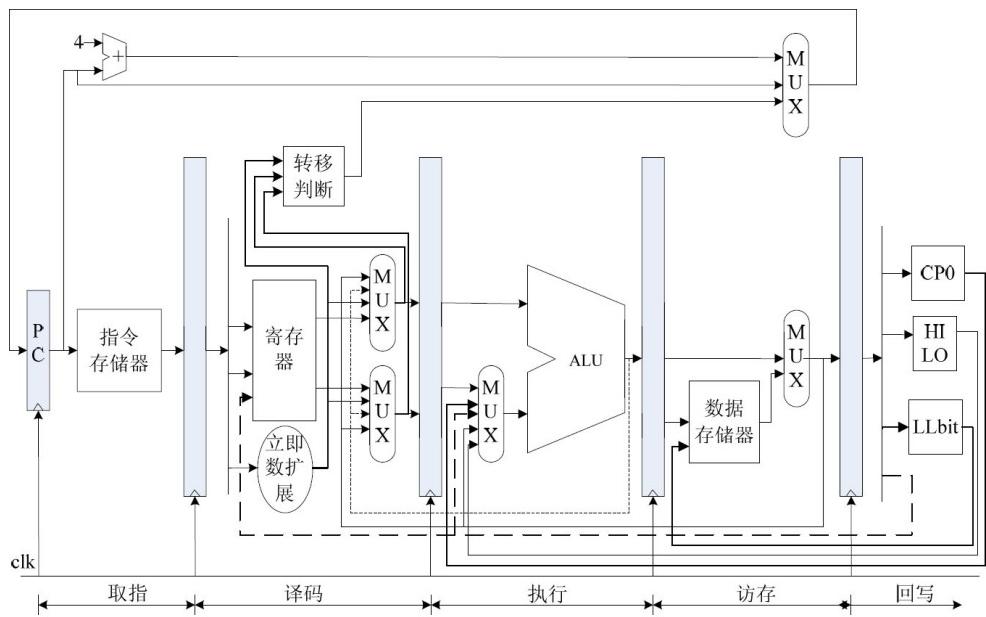


图10-3 添加协处理器CP0后的数据流图

10.5.3 系统结构的修改

为了实现对协处理器CP0的访问指令mtc0、mfc0，需要对系统结构进行如图10-4所示的修改。从图中观察，似乎增加了不少接口，但实际上很好理解。

如果是读取CP0中寄存器的指令mfc0，那么在执行阶段的EX模块会通过接口cp0_reg_read_addr_o输出要读取的CP0中寄存器的地址，该接口直接与CP0模块相连，正是图10-4中加粗的连接线。CP0模块通过data_o接口送出相应的数据，送出的数据通过EX模块的接口cp0_reg_data_i进入EX模块，正是图10-4中加粗的虚线。

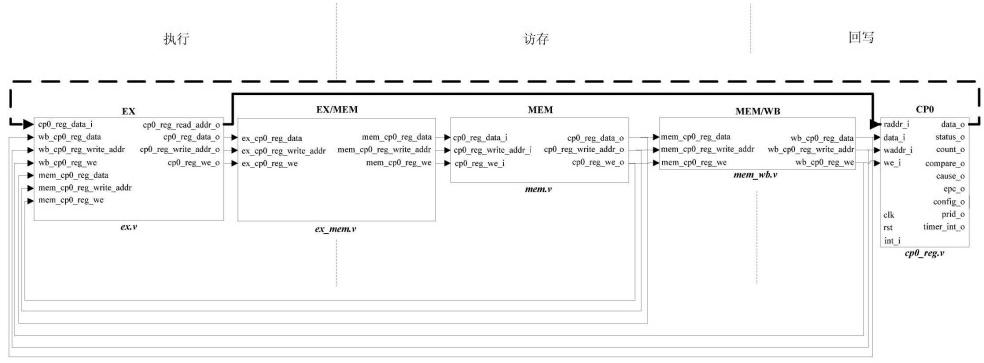


图10-4 为实现协处理器访问指令而对系统结构做的修改

如果是修改CP0中寄存器的指令mtc0，那么会在执行阶段的EX模块通过接口cp0_reg_we_o送出写信号，通过接口cp0_reg_write_addr_o送出要写的CP0中寄存器的地址，通过接口cp0_reg_data_o送出要写入的值，这些信息最终都传递到回写阶段，分别送入CP0模块的接口we_i、waddr_i、data_i，从而达到修改CP0中指定寄存器的目的。

需要特别说明一点，对于读取CP0中寄存器的指令mfc0，EX模块从CP0模块中读取的值可能不是最新的值，因为此时处于流水线访存、回写阶段的指令可能是mtc0，也就是说可能会修改CP0中的寄存器，读者朋友读到这里应该会心一笑，是的，这个问题，在第4章考虑流水线相关的时候遇到过，在第6章读/写HI、LO寄存器的时候遇到过，在第9章读/写LLbit寄存器的时候也遇到过，解决方法都是一样的——数据前推，此处也不例外，将访存、回写阶段对CP0中寄存器的写信息前推到执行阶段的EX模块，由EX模块判断得到最新的值。这也就是图10-4中，MEM模块、MEM/WB模块的输出会回送到EX模块的原因。

此外，图10-4中，CP0模块的各个寄存器输出接口暂时没有使用，在第11章实现异常处理的时候会使用到。

还需要注意的是，图10-4中，CP0模块的输入int_i是OpenMIPS处理器的输入，CP0模块的输出timer_int_o是OpenMIPS处理器的输出。由此，得到添加协处理器CP0后的OpenMIPS处理器接口示意图如图10-5所示。增加的接口如表10-12所示。

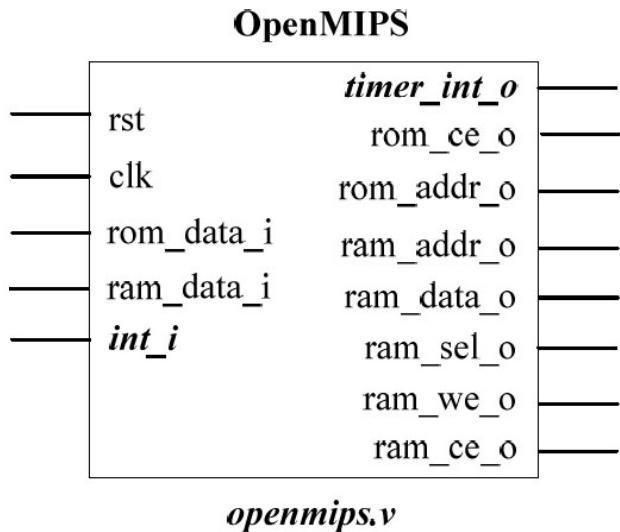


图10-5 添加协处理器CP0后的OpenMIPS处理器接口图

表10-12 OpenMIPS处理器新增加接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	int_i	6	输入	6个外部硬件中断输入
2	timer_int_o	1	输出	是否有定时中断发生

10.6 修改OpenMIPS以实现协处理器访问指令

10.6.1 修改译码阶段

译码阶段需要修改ID模块，在其中添加对mtc0、mfc0指令的译码。主要修改的代码如下所示，完整代码请参考本书附带光盘Code\Chapter10目录下的id.v文件。

```
module id(
    .....
);

    .....

always @ (*) begin
    if (rst == `RstEnable) begin
        .....
    end else begin
        .....
        if(inst_i[31:21] == 11'b010000000000 &&
           inst_i[10:0] == 11'b000000000000) //是mfc0指令
            begin
                aluop_o      <= `EXE_MFC0_OP;
                alusel_o     <= `EXE_RES_MOVE;
                wd_o         <= inst_i[20:16];
            end
    end
end
```

```
wreg_o      <= `WriteEnable;
instvalid   <= `InstValid;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b0;
end else if(inst_i[31:21] == 11'b01000000100 &&
            inst_i[10:0] == 11'b00000000000) //
```

是mtc0指令

```
begin
    aluop_o      <= `EXE_MTC0_OP;
    alusel_o     <= `EXE_RES_NOP;
    wreg_o       <= `WriteDisable;
    instvalid   <= `InstValid;
    reg1_read_o <= 1'b1;
    reg1_addr_o <= inst_i[20:16];
    reg2_read_o <= 1'b0;
end
end
.....
endmodule
```

从图10-2中可以发现这2条指令的格式与之前已实现的指令都不同，单独依据指令码无法区分这2条指令，所以此处直接通过指令第21~31bit的值判断区分mfc0、mtc0指令，另外，从图10-2中还可知，这2条指令要求第0~10bit都为0（对OpenMIPS而言，其中sel域也为0）。

译码工作主要是确定要写的目的寄存器、要读取的寄存器、要执行的运算等三个方面的信息。以下分别解释这两条指令的译码工作。

(1) mfc0指令

- 要写的目的寄存器：mfc0指令需要将读取的CP0中寄存器的值写入目的寄存器，所以设置wreg_o为WriteEnable，同时，参考图10-2可知，要写的目的寄存器是指令中的16-20bit，正是指令中rt的值，所以设置wd_o为inst[20:16]。
- 要读取的寄存器：mfc0指令不需要读取通用寄存器，所以设置reg1_read_o、reg2_read_o都为0。
- 要执行的运算：设置alusel_o为EXE_RES_MOVE，表示mfc0指令也是一种移动运算，设置aluop_o为EXE_MFC0_OP，表示运算子类型是mfc0。

(2) mtc0指令

- 要写的目的寄存器：mtc0指令不需要写通用寄存器，所以设置wreg_o为WriteDisable。
- 要读取的寄存器：mtc0指令需要读取通用寄存器，所以设置reg1_read_o为1，表示通过Regfile模块的读端口1读取数据，读取地址reg1_addr_o是指令中的第16~20bit，正是图10-2中

的rt的值，所以最终译码阶段的输出reg1_o就是地址为rt的通用寄存器的值。

- 要执行的运算：设置alusel_o为EXE_RES_MOVE，表示mtc0指令也是一种移动运算，设置aluop_o为EXE_MTC0_OP，表示运算子类型是mtc0。

10.6.2 修改执行阶段

1. 修改EX模块

参考图10-4可知，EX模块需要增加一些接口，新增接口的描述如表10-13所示。

表10-13 EX模块新增加接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	cp0_reg_data_i	32	输入	从 CP0 模块读取的指定寄存器的值
2	mem_cp0_reg_we	1	输入	访存阶段的指令是否要写 CP0 中的寄存器
3	mem_cp0_reg_write_addr	5	输入	访存阶段的指令要写的 CP0 中寄存器的地址
4	mem_cp0_reg_data	32	输入	访存阶段的指令要写入 CP0 中寄存器的数据
5	wb_cp0_reg_we	1	输入	回写阶段的指令是否要写 CP0 中的寄存器
6	wb_cp0_reg_write_addr	5	输入	回写阶段的指令要写的 CP0 中寄存器的地址

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
7	wb_cp0_reg_data	32	输入	回写阶段的指令要写入 CP0 中寄存器的数据
8	cp0_reg_read_addr_o	5	输出	执行阶段的指令要读取的 CP0 中寄存器的地址
9	cp0_reg_we_o	1	输出	执行阶段的指令是否要写 CP0 中的寄存器
10	cp0_reg_write_addr_o	5	输出	执行阶段的指令要写的 CP0 中寄存器的地址
11	cp0_reg_data_o	32	输出	执行阶段的指令要写入 CP0 中寄存器的数据

EX模块的代码主要修改如下，完整代码请参考本书附带光盘中 Code\Chapter10 目录下的ex.v文件。

```
module ex(
    . . .
    // 访存阶段的指令是否要写CP0中的寄存器，用来检测数据相关
    input wire                      mem_cp0_reg_we,
    input wire[4:0]                   mem_cp0_reg_write_addr,
    input wire[`RegBus]               mem_cp0_reg_data,
```

```

// 回写阶段的指令是否要写CP0中的寄存器，也是用来检测数据相关
input wire                      wb_cp0_reg_we,
input wire[4:0]                  wb_cp0_reg_write_addr,
input wire[`RegBus]              wb_cp0_reg_data,

// 与CP0直接相连，用于读取其中指定寄存器的值
input wire[`RegBus]              cp0_reg_data_i,
output reg[4:0]                  cp0_reg_read_addr_o,

// 向流水线下一级传递，用于写CP0中的指定寄存器
output reg                      cp0_reg_we_o,
output reg[4:0]                  cp0_reg_write_addr_o,
output reg[`RegBus]              cp0_reg_data_o,

      .....

);

      .....

/*********************************************
**
***** 第一段： 获得CP0中指定寄存器的值 *****
*****
**/


always @ (*) begin
  if(rst == `RstEnable) begin

```

```

moveres <= `ZeroWord;
end else begin
    moveres <= `ZeroWord;
    case (aluop_i)
        `EXE_MFHI_OP: begin
            moveres <= HI;
        end
        `EXE_MFL0_OP: begin
            moveres <= L0;
        end
        `EXE_MOVZ_OP: begin
            moveres <= reg1_i;
        end
        `EXE_MOVN_OP: begin
            moveres <= reg1_i;
        end
    end
`EXE_MFC0_OP: begin
    //要从CP0中读取的寄存器的地址
    cp0_reg_read_addr_o <= inst_i[15:11];
    //读取到的CP0中指定寄存器的值
    moveres <= cp0_reg_data_i;
    //判断是否存在数据相关
    if( mem_cp0_reg_we == `WriteEnable &&
        mem_cp0_reg_write_addr == inst_i[15:11] )

```

```

begin
    moveres <= mem_cp0_reg_data;           //与访存阶段存
在数据相关

end else if( wb_cp0_reg_we == `WriteEnable &&
            wb_cp0_reg_write_addr ==
inst_i[15:11] )

begin
    moveres <= wb_cp0_reg_data;           //与回写阶段存
在数据相关

end

default : begin
    end
endcase
end
end

/*****
**
***** 第二段： 确定最终要写入目的寄存器的值
*****
***** */


```

```
always @ (*) begin

    . . .

    case ( alusel_i )
        `EXE_RES_LOGIC: begin
            wdata_o <= logicout;
        end
        `EXE_RES_SHIFT: begin
            wdata_o <= shiftres;
        end

        `EXE_RES_MOVE: begin
            wdata_o <= moveres;
        end

        `EXE_RES_ARITHMETIC: begin
            wdata_o <= arithmeticres;
        end
        `EXE_RES_MUL: begin
            wdata_o <= mulres[31:0];
        end
        `EXE_RES_JUMP_BRANCH: begin
```

```

        wdata_o <= link_address_i;
    end
    default: begin
        wdata_o <= `ZeroWord;
    end
endcase
end

.....

```

```

//****************************************************************************
**  

***** 第三段：给出mtc0指令的执行结果 *****  

*****  

**/  


```

```

always @ (*) begin
    if(rst == `RstEnable) begin
        cp0_reg_write_addr_o <= 5'b00000;
        cp0_reg_we_o          <= `WriteDisable;
        cp0_reg_data_o         <= `ZeroWord;

```

```

end else if(aluop_i == `EXE_MTC0_OP) begin //是mtc0指令
    cp0_reg_write_addr_o <= inst_i[15:11];
    cp0_reg_we_o          <= `WriteEnable;
    cp0_reg_data_o         <= reg1_i;

```

```
    end else begin
        cp0_reg_write_addr_o <= 5'b00000;
        cp0_reg_we_o          <= `WriteDisable;
        cp0_reg_data_o         <= `ZeroWord;
    end
end

endmodule
```

上述代码可以分为三段理解。前两段与mfc0指令有关，最后一段与mtc0指令有关。

第一段：获得CP0中指定寄存器的值。首先通过cp0_reg_read_addr_o向CP0模块送出要读取的CP0中寄存器的地址，从10.3节CP0的实现代码可知，读取操作是组合逻辑，所以可以在一个时钟周期内给出相应数据，通过cp0_reg_data_i接口送入EX模块，并赋给变量moveres，但是需要注意此时的moveres并不一定是CP0中指定寄存器的最新值，还要判断是否存在数据相关。所以下面接着判断访存阶段的指令是否要写CP0中的寄存器，而且要写的是同一个寄存器，如果是，那么将访存阶段要写入的值，作为CP0中指定寄存器的最新值，反之，继续判断回写阶段的指令是否要写CP0，而且要写的是同一个寄存器，如果是，那么将回写阶段要写入的值，作为CP0中指定寄存器的最新值。

第二段：依据指令的运算类型，确定最终要写入目的寄存器的值，由于之前在译码阶段，将指令 mfc0 的运算类型设置为 EXE_RES_MOVE，所以会将 moveres 的值作为要写入目的寄存器的值，此处的 moveres 是在第一段代码中获取的。

第三段：如果是 mtc0 指令，那么给出对 CP0 中寄存器的写信息：设置写操作信号 cp0_reg_we_o 为 WriteEnable、写入地址为指令中第 11 ~ 15bit 的值、写入的值就是译码阶段传递过来的 reg1_i 的值，参考 10.6.1 节译码阶段可知，该值正是地址为 rt 的通用寄存器的值。

2. 修改EX/MEM模块

EX/MEM 模块会将 EX 模块得到的对 CP0 中寄存器的写信息向流水线下一级传递，参考图 10-4 可知，EX/MEM 增加了部分接口，新增接口的作用如表 10-14 所示。

表 10-14 EX/MEM 模块新增接口的描述

序号	接口名	宽度 (bit)	输入/输出	作用
1	ex_cp0_reg_we	1	输入	执行阶段的指令是否要写 CP0 中的寄存器

续表

序号	接口名	宽度 (bit)	输入/输出	作用
2	ex_cp0_reg_write_addr	5	输入	执行阶段的指令要写的 CP0 中寄存器的地址
3	ex_cp0_reg_data	32	输入	执行阶段的指令要写入 CP0 中寄存器的数据
4	mem_cp0_reg_we	1	输出	访存阶段的指令是否要写 CP0 中的寄存器
5	mem_cp0_reg_write_addr	5	输出	访存阶段的指令要写的 CP0 中寄存器的地址
6	mem_cp0_reg_data	32	输出	访存阶段的指令要写入 CP0 中寄存器的数据

EX/MEM模块的代码主要修改如下。完整代码可以参考本书附带光盘中Code\Chapter10目录下的ex_mem.v文件。

```
module ex_mem(  
    .....  
  
    // 新增输入接口  
    input wire ex_cp0_reg_we,  
    input wire[4:0] ex_cp0_reg_write_addr,  
    input wire[`RegBus] ex_cp0_reg_data,  
    .....  
  
    // 新增输出接口  
    output reg mem_cp0_reg_we,  
    output reg[4:0] mem_cp0_reg_write_addr,  
    output reg[`RegBus] mem_cp0_reg_data,  
    .....  
);  
  
always @ (posedge clk) begin  
    if(rst == `RstEnable) begin  
        .....  
        mem_cp0_reg_we      <= `WriteDisable;
```

```

    mem_cp0_reg_write_addr <= 5'b00000;
    mem_cp0_reg_data          <= `ZeroWord;
end else if(stall[3] == `Stop && stall[4] == `NoStop)
begin
    .....
    mem_cp0_reg_we           <= `WriteDisable;
    mem_cp0_reg_write_addr <= 5'b00000;
    mem_cp0_reg_data          <= `ZeroWord;
end else if(stall[3] == `NoStop) begin
    .....

```

// 在执行阶段没有暂停的时候，将对CP0中寄存器的写信息传递到访存阶段

```

mem_cp0_reg_we      <= ex_cp0_reg_we;
mem_cp0_reg_write_addr      <=
ex_cp0_reg_write_addr;
mem_cp0_reg_data          <= ex_cp0_reg_data;

```

.....

10.6.3 修改访存阶段

1. 修改MEM模块

MEM模块会将执行阶段传递过来的，对CP0中寄存器的写信息继续传递到流水线下一级，参考图10-4可知，MEM模块增加了部分接口，新增接口的作用如表10-15所示。

表10-15 MEM模块新增接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	cp0_reg_we_i	1	输入	访存阶段的指令是否要写 CP0 中的寄存器
2	cp0_reg_write_addr_i	5	输入	访存阶段的指令要写的 CP0 中寄存器的地址
3	cp0_reg_data_i	32	输入	访存阶段的指令要写入 CP0 中寄存器的数据
4	cp0_reg_we_o	1	输出	访存阶段的指令最终是否要写 CP0 中的寄存器
5	cp0_reg_write_addr_o	5	输出	访存阶段的指令最终要写的 CP0 中寄存器的地址
6	cp0_reg_data_o	32	输出	访存阶段的指令最终要写入 CP0 中寄存器的数据

MEM模块的代码主要修改如下，只是简单地将对CP0中寄存器的写信息传递到流水线下一级。完整代码可以参考本书光盘中Code\Chapter10目录下的mem.v文件。

```
module mem(  
    . . . . .  
    input wire cp0_reg_we_i,  
    input wire[4:0] cp0_reg_write_addr_i,  
    input wire[`RegBus] cp0_reg_data_i,
```

```
....  
  
output reg cp0_reg_we_o,  
output reg[4:0] cp0_reg_write_addr_o,  
output reg[`RegBus] cp0_reg_data_o,  
  
....  
  
);  
  
....  
  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        ....  
        cp0_reg_we_o <= `WriteDisable;  
        cp0_reg_write_addr_o <= 5'b00000;  
        cp0_reg_data_o <= `Zeroword;  
    end else begin  
        ....  
  
    // 将对CP0中寄存器的写信息传递到流水线下一级  
  
    cp0_reg_we_o <= cp0_reg_we_i;  
    cp0_reg_write_addr_o <= cp0_reg_write_addr_i;  
    cp0_reg_data_o <= cp0_reg_data_i;
```

.....

2. 修改MEM/WB模块

MEM/WB模块会将MEM模块传递过来的，对CP0中寄存器的写信息传递到回写阶段，参考图10-4可知，MEM/WB模块增加了部分接口，新增接口的作用如表10-16所示。

表10-16 MEM/WB模块新增的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	mem_cp0_reg_we	1	输入	访存阶段的指令是否要写 CP0 中的寄存器
2	mem_cp0_reg_write_addr	5	输入	访存阶段的指令要写的 CP0 中寄存器的地址
3	mem_cp0_reg_data	32	输入	访存阶段的指令要写入 CP0 中寄存器的数据

续表

序号	接口名	宽度(bit)	输入/输出	作用
4	wb_cp0_reg_we	1	输出	回写阶段的指令是否要写 CP0 中的寄存器
5	wb_cp0_reg_write_addr	5	输出	回写阶段的指令要写的 CP0 中寄存器的地址
6	wb_cp0_reg_data	32	输出	回写阶段的指令要写入 CP0 中寄存器的数据

MEM/WB模块的代码主要修改如下。完整代码可以参考本书附带光盘中Code\Chapter10目录下的mem_wb.v文件。

```
module mem_wb(
```

```

      .....

      input wire          mem_cp0_reg_we,
      input wire[4:0]      mem_cp0_reg_write_addr,
      input wire[`RegBus]   mem_cp0_reg_data,

      .....

      output reg           wb_cp0_reg_we,
      output reg[4:0]       wb_cp0_reg_write_addr,
      output reg[`RegBus]   wb_cp0_reg_data

);

always @ (posedge clk) begin
  if(rst == `RstEnable) begin
    .....
    wb_cp0_reg_we        <= `WriteDisable;
    wb_cp0_reg_write_addr <= 5'b00000;
    wb_cp0_reg_data      <= `ZeroWord;
  end else if(stall[4] == `Stop && stall[5] == `NoStop)
begin
  .....
  wb_cp0_reg_we        <= `WriteDisable;
  wb_cp0_reg_write_addr <= 5'b00000;

```

```
    wb_cp0_reg_data      <= `ZeroWord;
end else if(stall[4] == `NoStop) begin
    ....
// 在访存阶段没有暂停时，将对CP0中寄存器的写信息传递到回写阶
段
wb_cp0_reg_we      <= mem_cp0_reg_we;
wb_cp0_reg_write_addr  <= mem_cp0_reg_write_addr;
wb_cp0_reg_data      <= mem_cp0_reg_data;
end
end
endmodule
```

10.6.4 修改OpenMIPS模块

因为修改了一些模块的接口，所以需要修改OpenMIPS模块，按照图10-4所示，将新增加的接口连接在一起。需要注意一点，OpenMIPS模块本身也增加了两个接口：int_i、timer_int_o。具体代码不在书中给

出，读者可以参考本书光附带光盘Code\Chapter10目录下的openmips.v文件。

10.7 测试程序

为了验证本章添加的协处理器CP0，以及协处理器访问指令mfc0、mtc0是否实现正确，编写测试程序如下，源文件是本书附带光盘中Code\Chapter10\AsmTest目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:
    ori $1,$0,0xf      # $1 = 0xf
    mtc0 $1,$11,0x0    # 将0xf写入CP0中的Compare寄存器

    lui $1,0x1000
    ori $1,$1,0x401   # $1 = 0x10000401
    mtc0 $1,$12,0x0    # 将0x10000401写入CP0中的Status寄存器
    mfc0 $2,$12,0x0    # 读Status寄存器，保存到寄存器$2，$2 =
0x10000401

_loop:
```

```
j _loop
nop
```

程序首先写Compare寄存器，使其值等于0xf。这样，第15个时钟周期，Count寄存器的值等于Compare寄存器的值，会输出时钟中断（timer_int_o为1）。程序接着将0x10000401写入Status寄存器，然后读出Status寄存器的值，保存到寄存器\$2，用以验证读出、写入CP0中寄存器是否正确。ModelSim仿真效果如图10-6所示，从中可知OpenMIPS正确实现了协处理器CP0，以及协处理器访问指令mfc0、mtc0。

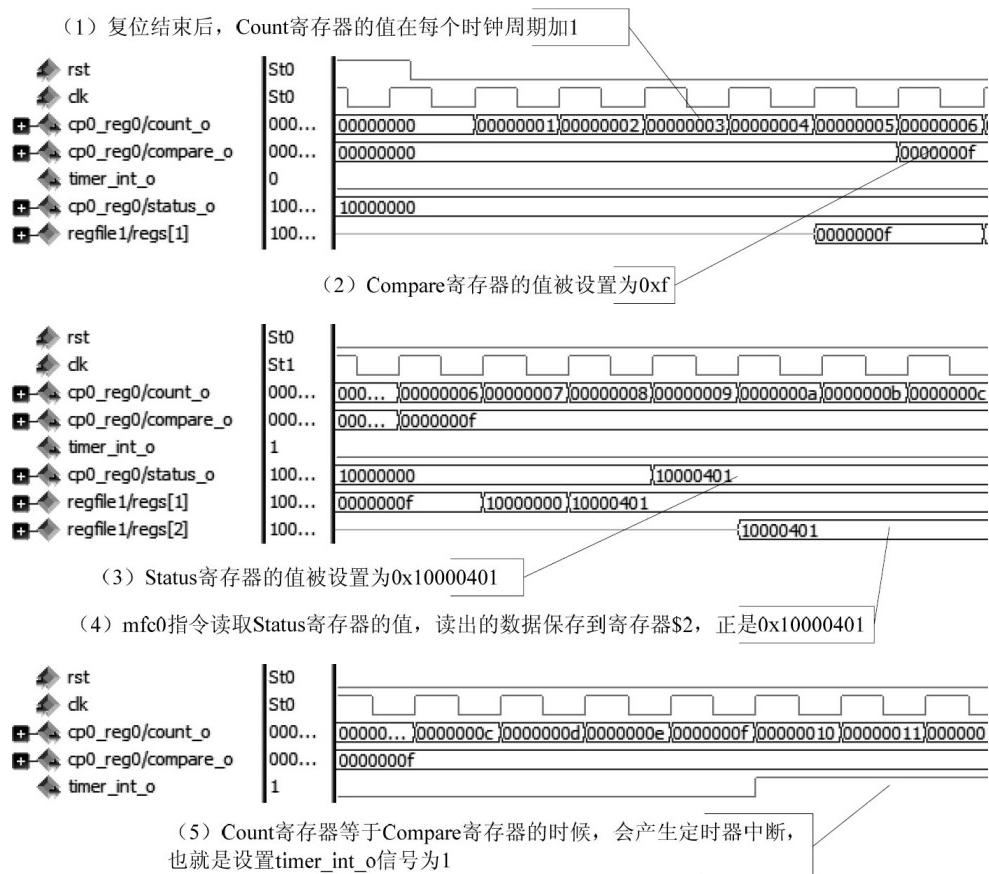


图10-6 ModelSim仿真测试效果

第11章 异常相关指令的实现

本章是实现教学版OpenMIPS处理器的最后一步，将实现异常相关指令。首先在11.1节介绍MIPS32架构中定义的异常类型，明确了OpenMIPS处理器能够处理的其中几种异常类型。随后在11.2节解释精确异常的概念，OpenMIPS处理器能够做到精确异常。11.3节介绍OpenMIPS处理器的异常处理过程，这个过程与MIPS32架构定义的异常处理过程稍微有些区别，为的是简单，易于理解。然后在11.4节对异常相关指令给出了说明，包括自陷指令、系统调用指令syscall、异常返回指令eret等。11.5节说明异常相关指令的实现思路，以及为了实现异常相关指令而对数据流图、系统结构作的修改。11.6节通过修改OpenMIPS的代码实现异常相关指令。11.7节再次修改了最小SOPC，将时钟中断输出接口与其中一个中断输入接口相连，这样OpenMIPS就可以响应时钟中断异常了。11.8节编写了3个测试程序，用于验证异常相关指令是否实现正确。

11.1 MIPS32架构中定义的异常类型

在MIPS32架构中，有一些事件要打断程序的正常执行流程，这些事件有中断（Interrupt）、陷阱（Trap）、系统调用（System Call）以及其他任何可以打断程序正常执行流程的情况，统称为异常。异常类型及其优先级如表11-1所示。读者朋友如果没有时间，可以只理解其中使用灰色背景标注的异常，OpenMIPS处理器也只实现了对这些异常的响应处理。

表11-1 MIPS32架构中定义的异常类型及其优先级

优先级	异常	描述
1	Reset	硬件复位
2	Soft Reset	在发生致命错误后对系统的复位，是软复位
3	DSS	Debug Single Step单步调试
4	DINT	Debug Interrupt调试中断
5	NMI	不可屏蔽的中断
6	Machine Check	发生在TLB入口多重匹配时
7	Interrupt	发生在8个中断之一被检测到时，包括6个外部硬件中断、2个软件中断
8	Deferred Watch	与观测点有关的异常
9	DIB	Debug Hardware Instruction Break Match，指令硬件断点和正在执行的指令相符合
10	WATCH	取指地址与观测寄存器中的地址相同时发生
11	AdEL	Fetch Address Align Error取指地

		址对齐异常
12	TLB Refill	指令TLB失靶
13	TLB Invalid	指令TLB无效
14	IBE	Instruction Fetch Bus Error 取指令总线错误
15	DBp	断点，执行了SDBBP指令
16	Sys	执行了系统调用指令syscall
	Bp	执行了break指令
	CpU	在协处理器不存在或不可用的情况下，执行了协处理器指令
	RI	无效指令
	Ov	算术操作指令add、addi、sub运算溢出
	Tr	执行了自陷指令
17	DDBL/DDBS	Data Address Break or Data Value Break on Store 存储过程中，数据地址断点或数据值断点
18	WATCH	数据地址与观测寄存器中的地

		址相同时发生
19	AdEL	加载数据的地址未对齐
	AdES	存储数据的地址未对齐
20	TLB Refill	数据TLB失靶
21	TLB Invalid	
22	TLB Mod	对不可写的TLB进行了写操作
23	DBE	加载存储总线错误
24	DDBL	Data Hardware Breakpoint matched in load data compare 加载的数据与硬件断点设置的数据相等

OpenMIPS处理器只实现对其中6种异常情况的处理，列举如下：

- 硬件复位；
- 中断（包含软中断、硬中断）；
- syscall系统调用；
- 无效指令；
- 溢出；
- 自陷指令引发的异常。

异常发生后，会进入异常处理例程进行具体的异常处理，处理结束后，返回到异常发生前的状态继续执行。在上面的6种异常中，硬件复位是一种特殊的异常，特殊之处在于不用从异常处理例程返回，所以不用考虑保护现场，也不用保存返回地址，OpenMIPS对硬件复位异常的处理方法是很简单的：全部寄存器清零，从地址0x0处取指执行，这实际也就是复位的过程。所以硬件复位异常的处理过程不再论述，本章只论述其余5种异常的处理过程。

11.2 精确异常

在MIPS的文档中经常会读到“精确异常”这个术语，OpenMIPS的实现蓝图中也设计为实现精确异常，本节将介绍精确异常的相关概念。

当一个异常发生后，系统的顺序执行会被中断，此时有若干条指令处于流水线上的不同阶段，处理器会转移到异常处理例程，异常处理结束后返回原程序继续执行，因为不希望异常处理例程破坏原程序的正常执行，所以对于异常发生时，流水线上没有执行完的指令，必须记住它处于流水线的哪一个阶段，以便异常处理结束后能恢复执行，这便是精确异常。

对于一个实现精确异常的处理器，在异常发生时，都会有一个被异常打断的指令，称为异常受害者（Exception Victim），也可称为发生异常的指令，该指令前面的所有指令都要被执行到流水线的最后一个阶段，也就是正常执行完成，但是该指令及该指令之后的指令都要被取消，就像从来没有执行过一样。如图11-1所示。第2条指令add在执行阶段发生溢出异常，在这种情况下，已经到达访存阶段的第1条指

令ori会继续执行完成，而第2、3、4条指令都会被取消，不会有影响处理器的情况发生，就像没有进过流水线一样。

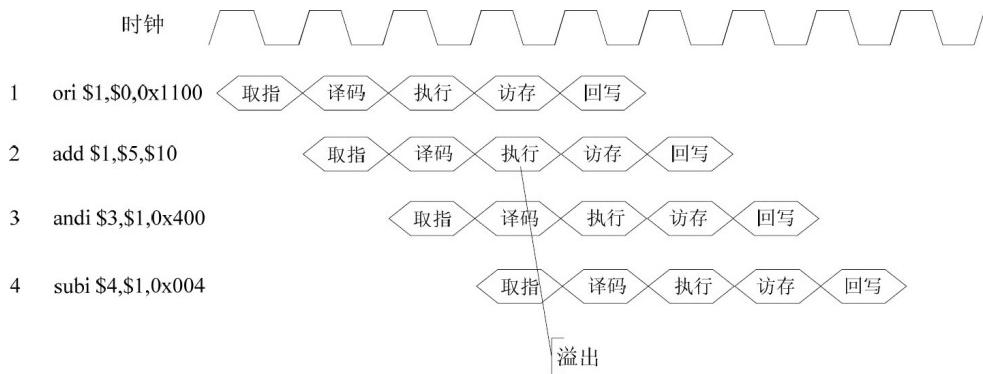


图11-1 精确异常示例

为了实现精确异常，必须要求异常发生的顺序与指令的顺序相同，在非流水线的处理器上，这一点是显然的，但是对于拥有流水线的处理器，就会有些复杂。在流水线处理器上，异常会在流水线的不同阶段发生，带来潜在的问题。比如：以图11-2为例，加载指令lw会在流水线的访存阶段发生地址未对齐的异常（因为加载地址是0x3，指令lw要求加载地址的最后两位为00），该异常应该会在第4个时钟周期发生，而它的后一条指令di是无效指令（MIPS32架构并没有定义该指令，所以是无效指令），会在流水线的译码阶段引发无效指令异常，也就是在第3个时钟周期，而此时上一条加载指令lw还处于执行阶段，没有进入访存阶段，所以先发生的异常就是无效指令异常。从而不满足异常发生的顺序与指令的顺序相同这一要求。

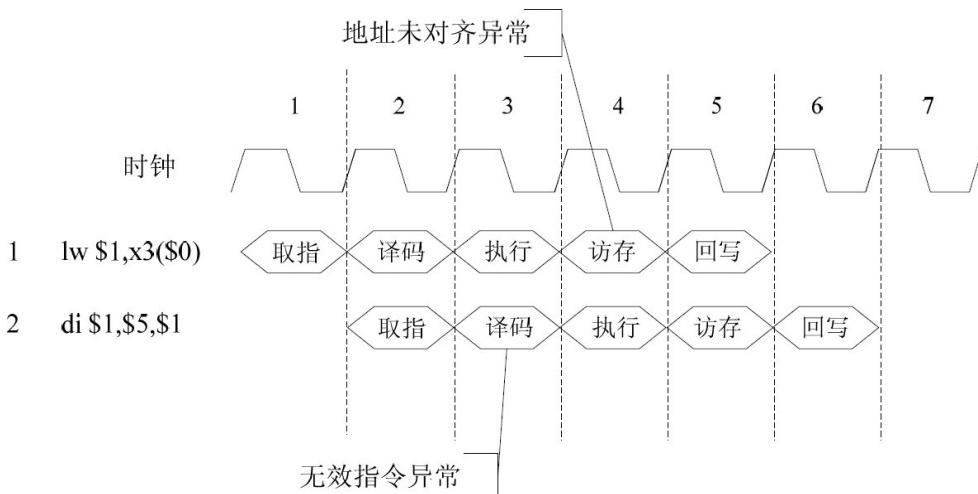


图11-2 在流水线处理器中，异常发生的顺序与指令的顺序不一定相同

为了避免上述情况，先发生的异常并不立即处理，异常事件只是被标记，并继续运行流水线。在大多数处理器中，会设计一个特殊的流水线阶段，专门用于处理异常。如果某一条指令的异常事件到达了流水线的这个阶段，那么会进行异常处理，并且当前处于流水线其余阶段的指令的异常事件都会被忽略。还是以图11-2为例，假设处理器会在访存阶段处理异常情况，那么di指令虽然在第3个时钟周期发生了异常，但是并不处理，只是保存一个异常标记，等到第5个时钟周期该指令进入访存阶段时再处理。但是，在第4个时钟周期，上一条指令lw进入了访存阶段，并且发生了地址未对齐异常，因为已经处于访存阶段了，所以会处理该异常，而包括无效指令异常在内的流水线其余阶段的异常都被忽略。

通过以上方法就可以在流水线处理器中实现“按指令执行的顺序处理异常，而不是按异常发生的顺序处理异常”。

11.3 异常处理过程

当检测到异常发生后，处理器会执行一系列动作以处理异常，不同处理器的处理过程也不同，OpenMIPS处理器的处理过程如下。

(1) 检测CP0中Status寄存器的EXL字段，分两种情况。

- 如果EXL为1，表示当前已经处于异常处理过程中了，此时，如果当前发生的异常类型是中断，那么不处理，忽略该异常，因为在异常处理过程中会禁止中断。如果当前发生的异常类型不是中断，那么将异常原因保存到CP0中Cause寄存器的ExcCode字段，转到步骤（4）。
- 如果EXL为0，那么将异常原因保存到CP0中Cause寄存器的ExcCode字段，进入步骤（2）。

(2) 检查发生异常的指令是否在延迟槽中，如果在延迟槽中，那么设置EPC寄存器的值为该指令的地址减4，同时设置Cause寄存器的BD字段为1，反之，设置EPC寄存器的值就为该指令的地址，同时设置Cause寄存器的BD字段为0。

(3) 设置Status寄存器的EXL字段为1，表示进入异常处理过程，禁止中断。

(4) 处理器转移到事先定义好的一个地址，在那个地址中往往有异常处理例程，在其中进行异常处理，这个地址称为异常处理例程入口地址。OpenMIPS定义的异常处理例程入口地址如表11-2所示。此处对系统调用、无效指令、溢出、自陷这四类异常都设置为相同的处理例程入口地址，当然也可以设置为不同的地址，读者在阅读完本章后，将学会如何根据情况自行修改。

以上就是OpenMIPS处理器在检测到异常发生后的处理过程，可以使用图11-3描述。熟悉MIPS32架构中异常处理过程的读者可能会注意到上述过程与MIPS32架构中的异常处理过程只有一点不同，那就是对异常处理例程入口地址的规定不同。OpenMIPS处理器由于没有MMU，所以将异常处理例程入口地址都放在低地址空间。

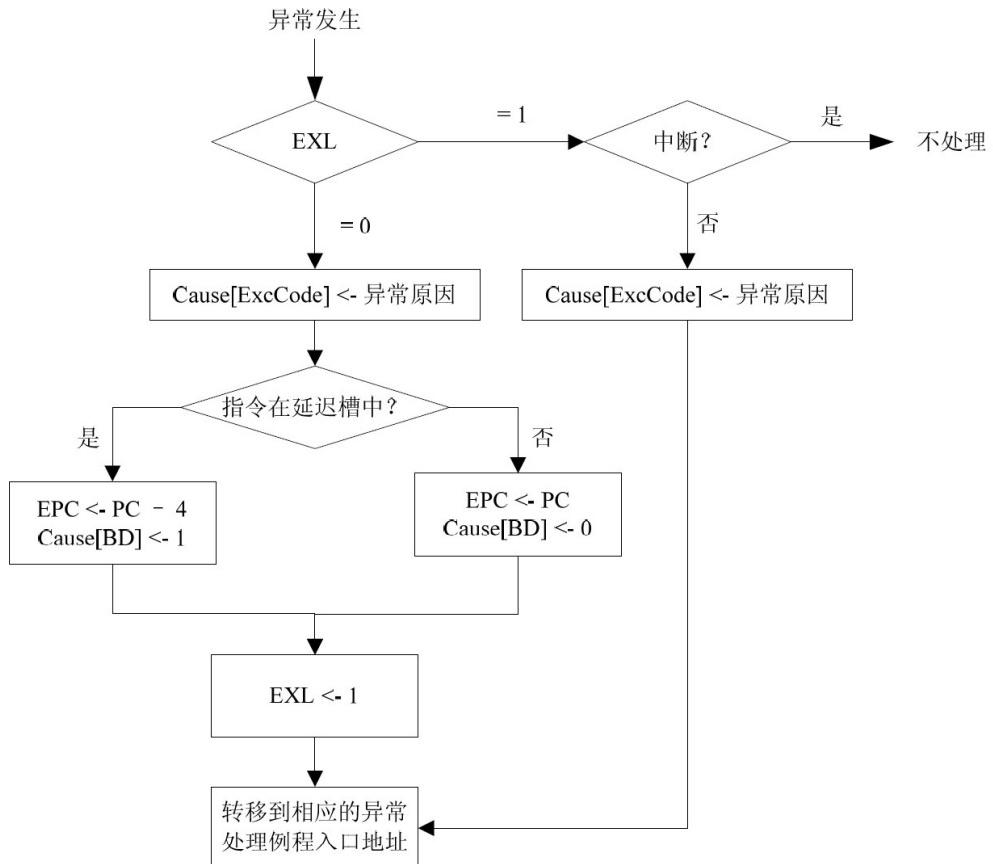


图11-3 OpenMIPS处理器的异常处理过程

在异常处理例程中会进行具体的异常处理，处理结束后，需要返回到异常发生前的状态继续执行。MIPS32指令集提供了异常返回指令`eret`来完成此项工作。`eret`指令既要清除Status寄存器的EXL字段，从而使能中断，还要将EPC寄存器保存的地址恢复到PC中，从而返回到异常发生处继续执行。

表11-2 OpenMIPS处理器定义的异常处理例程入口地址

异常类型	处理例程地址	引起异常的条件
中断 (Interrupt)	0x20	硬件或软件中断
系统调用Sys	0x40	执行了系统调用指令syscall
无效指令RI	0x40	当前指令是OpenMIPS不支持的指令
溢出Ov	0x40	算术操作指令add、addi、sub运算溢出
自陷Tr	0x40	执行了自陷指令

读者可能会注意到这个问题：如果发生异常的指令在延迟槽中，那么保存到寄存器EPC的值是PC-4，如果发生异常的指令不在延迟槽中，那么保存到寄存器EPC的值是PC，为何会有这种区别呢？

这是因为在引入延迟槽之前，处理器执行转移指令的顺序是。

转移指令->转移目标地址的指令

引入延迟槽之后，处理器执行转移指令的顺序是。

转移指令->延迟槽指令->转移目标地址的指令

在中间插入了延迟槽指令，当延迟槽中的指令发生异常时，如果在寄存器EPC中保存延迟槽指令的地址，那么从异常处理例程返回时，将回到延迟槽指令的地址处，重新执行的指令顺序将是。

延迟槽指令->延迟槽指令的下一条指令

可见没有发生转移，这样就不是被打断之前的指令顺序，所以，为了恢复原来的指令顺序，在这里将延迟槽之前的转移指令的地址保存到寄存器EPC中，也就是PC-4。

11.4 异常相关指令介绍

MIPS32指令集架构中定义的异常相关指令包括：自陷指令、系统调用指令syscall、异常返回指令eret，下面分别介绍。

11.4.1 自陷指令

自陷指令有12条，按照指令中是否包含立即数，可以分为两类。

1. 不包含立即数的自陷指令

不包含立即数的自陷指令有6条，指令格式如图11-4所示。

	31	26 25	21 20	16 15	6 5	0	
SPECIAL 000000		rs	rt	code	TEQ 110100		teq指令
SPECIAL 000000		rs	rt	code	TGE 110000		tge指令
SPECIAL 000000		rs	rt	code	TGEU 110001		tgeu指令
SPECIAL 000000		rs	rt	code	TLT 110010		tlt指令
SPECIAL 000000		rs	rt	code	TLTU 110011		tltu指令
SPECIAL 000000		rs	rt	code	TNE 110110		tne指令

图11-4 不包含立即数的自陷指令

从图11-4中可知，这6条自陷指令都是R类型指令，且指令码都是SPECIAL，可以依据第0~5bit功能码的值确定是哪一种指令。另外，指令的第6~15bit都是code字段，该字段在译码过程中没有作用，被忽略掉，但是软件可以利用这个字段保存一些信息。

- 当功能码为6'b110100时，是teq指令。

指令用法为： teq rs, rt。

指令作用为： if GPR[rs] = GPR[rt] then trap，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值进行比较，如果两者相等，那么引发自陷异常。

- 当功能码为6'b110000时，是tge指令。

指令用法为： tge rs, rt。

指令作用为： if GPR[rs] ≥ GPR[rt] then trap，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值作为有符号数进行比较，如

果前者大于等于后者，那么引发自陷异常。

- 当功能码为6'b110001时，是tgeu指令。

指令用法为：tgeu rs, rt。

指令作用为：if GPR[rs] \geq GPR[rt] then trap，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值作为无符号数进行比较，如果前者大于等于后者，那么引发自陷异常。

- 当功能码为6'b110010时，是tltr指令。

指令用法为：tltr rs, rt。

指令作用为：if GPR[rs] $<$ GPR[rt] then trap，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值作为有符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当功能码为6'b110011时，是tltru指令。

指令用法为：tltru rs, rt。

指令作用为：if GPR[rs] $<$ GPR[rt] then trap，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值作为无符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当功能码为6'b110110时，是tne指令。

指令用法为：tne rs, rt。

指令作用为：if GPR[rs] ≠ GPR[rt] then trap，将地址为rs的通用寄存器的值，与地址为rt的通用寄存器的值进行比较，如果两者不相等，那么引发自陷异常。

2. 包含立即数的自陷指令

包含立即数的自陷指令也有6条，指令格式如图11-5所示。

31	26 25	21 20	16 15	0	
REGIMM 000001	rs	TEQI 01100	immediate		teqi指令
REGIMM 000001	rs	TGEI 01000	immediate		tgei指令
REGIMM 000001	rs	TGEIU 01001	immediate		tgeiu指令
REGIMM 000001	rs	TLTI 01010	immediate		tlti指令
REGIMM 000001	rs	TLTIU 01011	immediate		tltiu指令
REGIMM 000001	rs	TNEI 01110	immediate		tnei指令

图11-5 包含立即数的自陷指令

从图11-5可知，这6条自陷指令都是I类型指令，且指令码都是REGIMM，可以依据第16~20bit的值确定是哪一种指令。这6条自陷指令与之前不包含立即数的6条自陷指令的区别在于，此处的自陷判断条件不再是两个寄存器的比较结果，而是一个寄存器与指令中立即数的比较结果。

- 当第16~20bit的值为5'b01100时，表示是teqi指令。

指令用法为：teqi rs, immediate。

指令作用为： if GPR[rs] = sign_extended(immediate) then trap， 将地址为rs的通用寄存器的值， 与指令中16位立即数符号扩展至32位后的值进行比较， 如果两者相等， 那么引发自陷异常。

- 当第16~20bit的值为5'b01000时， 表示是tgei指令。

指令用法为： tgei rs, immediate。

指令作用为： if GPR[rs] ≥ sign_extended(immediate) then trap， 将地址为rs的通用寄存器的值， 与指令中16位立即数符号扩展至32位后的值作为有符号数进行比较， 如果前者大于等于后者， 那么引发自陷异常。

- 当第16~20bit的值为5'b01001时， 表示是tgeiu指令。

指令用法为： tgeiu rs, immediate。

指令作用为： if GPR[rs] ≥ sign_extended(immediate) then trap， 将地址为rs的通用寄存器的值， 与指令中16位立即数符号扩展至32位后的值作为无符号数进行比较， 如果前者大于等于后者， 那么引发自陷异常。

- 当16-20bit的值为5'b01010时， 表示是tlti指令。

指令用法为： tlti rs, immediate。

指令作用为： if GPR[rs] < sign_extended(immediate) then trap， 将地址为rs的通用寄存器的值， 与指令中16位立即数符号扩展至32位后的值作为有符号数进行比较， 如果前者小于后者， 那么引发自陷异常。

- 当16-20bit的值为5'b01011时，表示是tluiu指令。

指令用法为： tluiu rs, immediate。

指令作用为： if GPR[rs] < sign_extended(immediate) then trap， 将地址为rs的通用寄存器的值，与指令中16位立即数符号扩展至32位后的值作为无符号数进行比较，如果前者小于后者，那么引发自陷异常。

- 当16-20bit的值为5'b01110时，表示是tnei指令。

指令用法为： tnei rs, immediate。

指令作用为： if GPR[rs] ≠ sign_extended(immediate) then trap， 将地址为rs的通用寄存器的值，与指令中16位立即数符号扩展至32位后的值进行比较，如果两者不相等，那么引发自陷异常。

11.4.2 系统调用指令syscall

系统调用指令syscall的格式如图11-6所示。



图11-6 syscall指令的格式

从图11-6可知，syscall指令的指令码是SPECIAL，可以依据第0~5bit功能码是否是6'b001100，进而判断是否是syscall指令。另外，指令中的第6~25bit是code字段，该部分在译码过程中没有作用，被忽略掉，但是软件可以利用这个字段保存一些信息。

指令用法为：syscall。

指令作用为：引发系统调用异常。MIPS32架构定义了处理器的两种工作模式：用户模式、内核模式，前一种是受限模式，有些操作无法进行，大多用于用户的应用程序，后者主要用于处理异常和具有优先权的操作系统函数，包括管理协处理器CP0和I/O等。用户模式下的程序为了执行一些在内核模式下才能进行的操作，可以调用syscall指令，，引发系统调用异常，进入异常处理例程，从而进入内核模式。用户模式、内核模式的状态标记是CP0中Status寄存器的UM字段。

OpenMIPS不区分用户模式、内核模式，所以没有使用Status寄存器的UM字段，也就是说，所有的操作都没有限制，但是为了兼容MIPS32指令集架构，还是实现了syscall指令。

11.4.3 异常返回指令eret

异常返回指令eret的格式如图11-7所示。

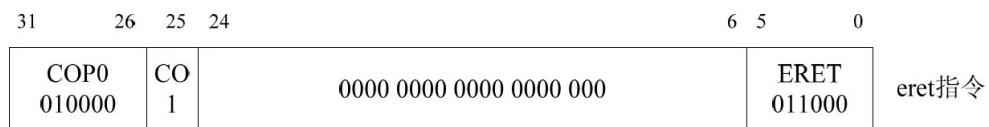


图11-7 异常返回指令eret的格式

从图11-7可知，eret的指令码是COP0，与上一章实现的指令mtc0、mfc0的指令码是一样的，但是第0~5bit功能码的值是6'b011000，而且第25bit为1，第6~24bit都为0。

指令用法为：eret

指令作用为：从异常处理例程返回，执行该指令，会进行如下操作。

- (1) 使EPC寄存器的值成为新的取指地址。
- (2) 设置Status寄存器的EXL字段为0，表示不再处于异常级。

11.5 异常处理实现思路

OpenMIPS处理器能够处理的几种异常，包括异常相关指令引起的异常（自陷指令、syscall、无效指令）、外部中断引起的异常，以及执行算术运算出现溢出引起的异常。这几种异常的处理过程都是一致的。另外，在实现异常处理的过程中，自然就实现了syscall、自陷指令等异常相关指令，所以本节以及11.6节都是从实现异常处理的角度讲解，而不是从异常相关指令的角度讲解。

还有一点，虽然异常返回指令eret不是引起异常的指令，但是eret的执行效果与异常的效果非常相似：取消随后所有指令的执行、转移到新的目标地址（对于eret指令而言，就是EPC中保存的地址）。所以eret指令的处理过程与异常处理过程也是类似的，可以称为“返回异常”。本章结合异常处理的实现过程一并介绍eret指令的实现过程。

11.5.1 实现思路

OpenMIPS异常处理的实现思路是：在流水线的各个阶段收集异常信息，并传递到流水线访存阶段，在访存阶段统一处理异常信息。流

流水线各个阶段需要收集的异常信息如下。

- 在流水线译码阶段判断是否是系统调用异常、是否是返回指令、无效指令。
- 在流水线执行阶段判断是否有自陷异常、溢出异常。
- 在流水线访存阶段检查是否有中断发生。

在流水线访存阶段，处理器将结合协处理器CP0中相关寄存器的值，判断异常是否需要处理，如果需要处理，那么转移到该异常对应的处理例程入口地址，清除流水线上除回写阶段外的全部信息（回写阶段的指令要继续执行，参考“精确异常”一节的描述），同时，修改协处理器CP0中相关寄存器的值。

如果是`eret`指令，那么转移到EPC寄存器保存的地址处，同时，也要清除流水线上除回写阶段外的全部信息，修改协处理器CP0中相关寄存器的值。

清除流水线上某个阶段的信息，实际就是将该阶段中的所有寄存器设置为初始值即可。

11.5.2 修改数据流图

添加异常处理后的数据流图如图11-8所示，相比图10-3，在访存阶段增加了异常判断模块，主要作用是依据从译码、执行阶段传递过来的信息，以及CP0中寄存器的值，判断是否要处理异常，如果要处理异常，那么按照异常类型给出新的指令地址送入PC。

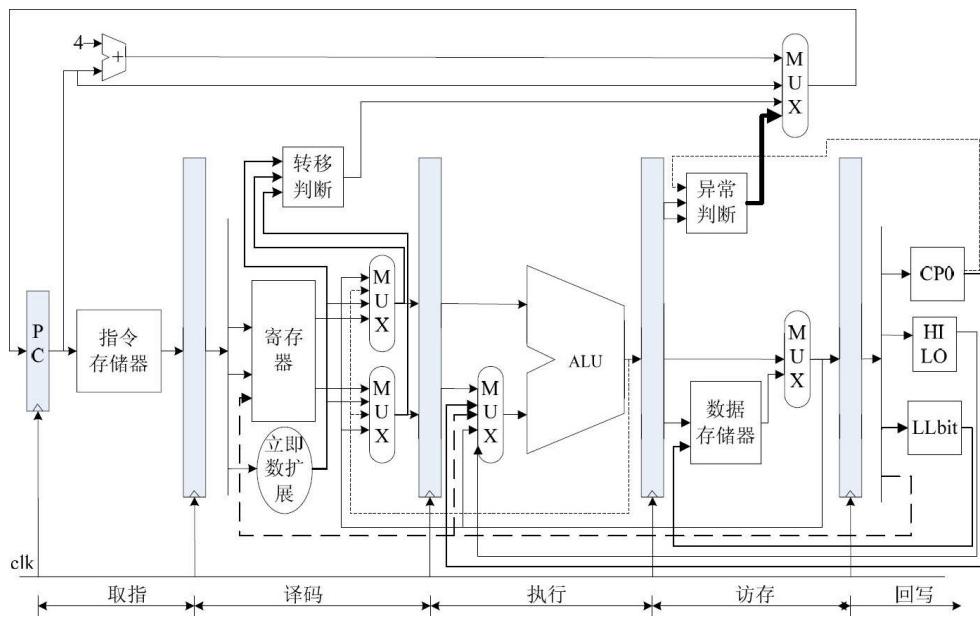


图11-8 添加异常处理后的数据流图

11.5.3 修改系统结构

为了实现异常处理，需要修改系统结构，添加部分接口，如图11-9所示。主要有如下几点说明。

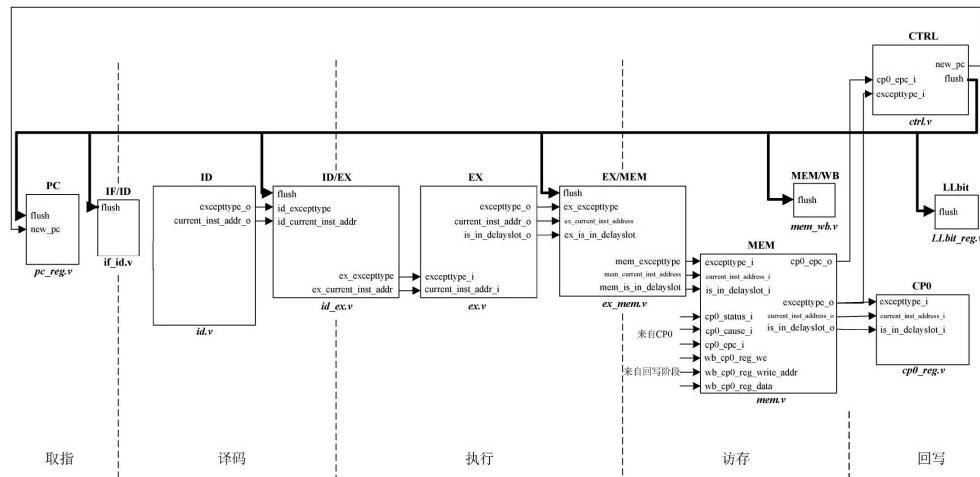


图11-9 为实现异常处理而对OpenMIPS系统结构所做的修改

(1) 流水线译码阶段ID模块会判断是否是系统调用指令syscall、异常返回指令eret、无效指令，将这些信息通过excepttype_o接口传递到执行阶段，同时，还将指令地址通过current_inst_addr_o接口传递到执行阶段。

(2) 流水线执行阶段EX模块会进一步判断是否有自陷异常，或者溢出异常。这些信息会融合到译码阶段给出的异常信息中（通过EX模块的excepttype_i接口传入），然后通过excepttype_o接口传递到访存阶段。同时，通过current_inst_addr_o接口将指令地址传递到访存阶段，通过is_in_delayslot_o接口指出指令是否位于延迟槽中，该信息也被传递到访存阶段。

(3) 流水线访存阶段MEM模块会依据传递过来的异常类型excepttype_i、Cause寄存器的值（通过cp0_cause_i接口输入）、Status寄存器的值（通过cp0_status_i接口输入），综合判断是否需要处理异常，如果需要处理，那么最终的异常类型会通过excepttype_o接口送入CTRL模块，CTRL模块据此给出异常处理入口地址（通过new_pc接口送至PC）。

(4) 如果要处理异常，那么还需要修改协处理器CP0中EPC、Status、Cause等寄存器的值，所以访存阶段给出的最终的异常类型还要通过excepttype_o接口送入CP0模块，同时送入的还有发生异常的指令是否在延迟槽中（通过is_in_delayslot_i接口送入）、发生异常的指令的地址（通过current_inst_address_o接口送入）。CP0模块依据这些信息修改相应寄存器的值。

(5) 如果要处理异常，那么还需要清除流水线上除回写阶段外的所有寄存器的值，CTRL模块通过送出flush信号实现此目的。从图11-9

可知，flush信号送到PC、IF/ID、ID/EX、EX/MEM、MEM/WB等模块，会将这些模块中的寄存器置为初始值。

(6) 在第9章实现ll、sc指令的时候引入了LLbit寄存器，当ll指令执行的时候会设置LLbit为1，当sc指令执行的时候，会检查该寄存器是否为1，如果为1就正常执行；如果为0，那么认为出现了干扰，不进行存储操作。出现干扰的原因之一就是在ll、sc指令之间产生了异常，所以在异常处理过程中会多进行一步操作，就是将LLbit寄存器置为0。这也是图11-9中LLbit模块也有flush信号输入的原因。

11.6 修改OpenMIPS以实现异常处理

11.6.1 修改取指阶段

1. 修改PC模块

从图11-9可知，PC模块会增加部分接口，如表11-3所示。

表11-3 PC模块增加的接口

序号	接口名	宽度(bit)	输入/输出	作用
1	flush	1	输入	流水线清除信号
2	new_pc	32	输入	异常处理例程入口地址

PC模块的代码修改如下，修改部分使用加粗、斜体强调。完整代码请参考本书附带光盘Code\Chapter11目录下的pc_reg.v文件。

```
module pc_reg(
    .....  

    input wire flush,  

    input wire[`RegBus] new_pc,  

    .....  

);  

    .....  

    always @ (posedge clk) begin  

        if (ce == `ChipDisable) begin  

            pc <= 32'h00000000;  

        end else begin  

if(flush == 1'b1) begin  

            // 输入信号flush为1表示异常发生，将从CTRL模块给出的异常处理  

            // 例程入口地址new_pc处取指执行  

            pc <= new_pc;  

        end  

    end  

);
```

```

        end else if(stall[0] == `NoStop) begin
            if(branch_flag_i == `Branch) begin
                pc <= branch_target_address_i;
            end else begin
                pc <= pc + 4'h4;
            end
        end
    end
endmodule

```

2. 修改IF/ID模块

从图11-9可知， IF/ID模块也要增加部分接口，如表11-4所示。

表11-4 IF/ID模块增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	flush	1	输入	流水线清除信号

IF/ID模块的代码修改如下，修改部分使用加粗、斜体强调。完整代码请参考本书附带光盘中Code\Chapter11目录下的if_id.v文件。

```
module if_id(
```

```
    ....
```

```
input wire      flush,  
      .....  
);  
  
always @ (posedge clk) begin  
    if (rst == `RstEnable) begin  
        id_pc    <= `ZeroWord;  
        id_inst <= `ZeroWord;  
  
    end else  
if(flush == 1'b1 ) begin  
        // flush为1表示异常发生，要清除流水线，  
        // 所以复位id_pc、id_inst寄存器的值  
        id_pc    <= `ZeroWord;  
        id_inst <= `ZeroWord;  
  
    end else if(stall[1] == `Stop && stall[2] == `NoStop)  
begin  
    id_pc    <= `ZeroWord;  
    id_inst <= `ZeroWord;
```

```

    end else if(stall[1] == `NoStop) begin
        id_pc    <= if_pc;
        id_inst <= if_inst;
    end
end

endmodule

```

11.6.2 修改译码阶段

1. 修改ID模块

从图11-9可知，译码阶段ID模块也要增加部分接口，如表11-5所示。

表11-5 ID模块增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	excepttype_o	32	输出	收集的异常信息
2	current_inst_addr_o	32	输出	译码阶段指令的地址

译码阶段ID模块要增加对自陷指令、系统调用指令syscall、异常返回指令eret的译码过程，首先要确定是哪一种指令，确定指令的过程如图11-10所示。其中对eret指令的确定比较特别，是直接将指令与宏定义EXE_ERET比较，如果相等，那么就是eret指令。

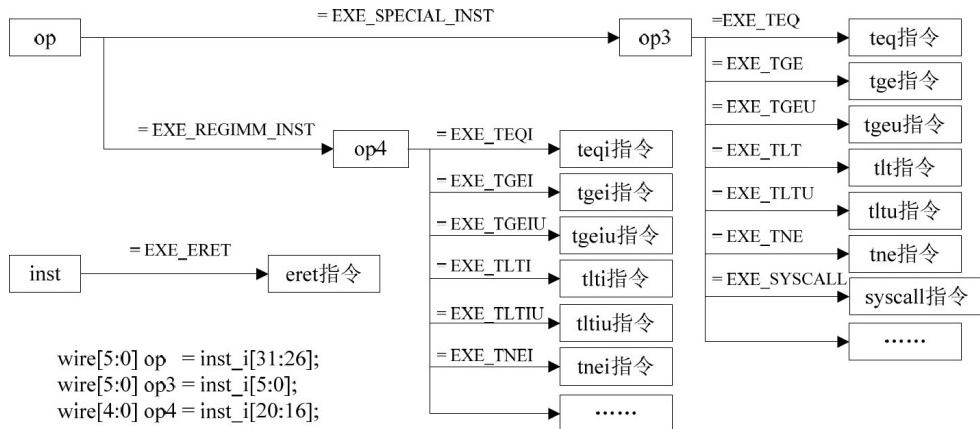


图11-10 确定自陷指令、syscall指令、eret指令的过程

其中涉及的宏定义如下，在本书附带光盘中Code\Chapter11目录下的defines.v文件中可以找到这些定义。

```

`define EXE_SYSCALL      6'b001100

`define EXE_TEQ          6'b110100
`define EXE_TEQI         5'b01100
`define EXE_TGE          6'b110000
`define EXE_TGEI         5'b01000
`define EXE_TGEIU        5'b01001
`define EXE_TGEU         6'b110001
`define EXE_TLT          6'b110010
`define EXE_TLTI         5'b01010
`define EXE_TLTIU        5'b01011
`define EXE_TLTU         6'b110011
`define EXE_TNE          6'b110110
`define EXE_TNEI         5'b01110

```

```
`define EXE_ERET      32'b01000010000000000000000000000000
```

此外还有一宏定义，会在异常处理过程中使用到，如下：

```
`define InstValid      1'b0
`define InstInvalid    1'b1
`define InDelaySlot    1'b1
`define NotInDelaySlot 1'b0
`define InterruptAssert 1'b1
`define InterruptNotAssert 1'b0
`define TrapAssert     1'b1
`define TrapNotAssert  1'b0
```

ID 模块的代码主要修改如下，完整代码位于本书光盘 Code\Chapter11 目录下的 id.v 文件。

```
module id(
    .....
    // 新增的输出接口
    output wire[31:0]      excepttype_o,
    output wire[`RegBus]    current_inst_address_o,
    .....
);

reg excepttype_is_syscall; // 是否是系统调用异常syscall
```

```
reg excepttype_is_eret;      // 是否是异常返回指令eret

.....



// excepttype_o的低8bit留给外部中断，第8bit表示是否是syscall指令引起的

// 系统调用异常，第9bit表示是否是无效指令引起的异常，第12bit表示是否是eret

// 指令，eret指令可以认为是一种特殊的异常—返回异常

assign excepttype_o = {19'b0,
excepttype_is_eret, 2'b0,
instvalid, excepttype_is_syscall,
8'b0};




// 输入信号pc_i就是当前处于译码阶段的指令的地址

assign current_inst_address_o = pc_i;






always @ (*) begin

if (rst == `RstEnable) begin

.....



end else begin

.....



aluop_o      <= `EXE_NOP_OP;
alusel_o     <= `EXE_RES_NOP;
```

```

        wd_o          <= inst_i[15:11];           //默认目的寄存器地
址wd_o

        wreg_o        <= `WriteDisable;
        instinvalid   <= `InstInvalid;
        reg1_read_o   <= 1'b0;
        reg2_read_o   <= 1'b0;
        reg1_addr_o   <= inst_i[25:21];           // 默认的
reg1_addr_o

        reg2_addr_o   <= inst_i[20:16];           // 默认的
reg2_addr_o

        imm           <= `ZeroWord;
        .....
        ......



excepttype_is_syscall <= `False_v;           // 默认没有系统调用异常
excepttype_is_eret    <= `False_v;           // 默认不是eret指
令
instinvalid           <= `InstInvalid; // 默认是无效指令



case (op)
`EXE_SPECIAL_INST: begin
        .....
case (op3)

`EXE_TEQ: begin           // teq指令

```

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_SYSCALL_OP;
alusel_o     <= `EXE_RES_NOP;
reg1_read_o <= 1'b0;
reg2_read_o <= 1'b0;
instinvalid <= `InstInvalid;
end
```

`**EXE_TGE**: begin // **tge**指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_TGE_OP;
alusel_o     <= `EXE_RES_NOP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instinvalid <= `InstInvalid;
end
```

`**EXE_TGEU**: begin // **tgeu**指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_TGEU_OP;
alusel_o     <= `EXE_RES_NOP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instinvalid <= `InstValid;
end
```

`**EXE_TLT**: begin // tlt指令

```
wreg_o      <= `WriteDisable;
aluop_o      <= `EXE_TLT_OP;
alusel_o     <= `EXE_RES_NOP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instinvalid <= `InstValid;
end
```

`**EXE_TLTU**: begin // tltu指令

```
wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_TLTU_OP;
alusel_o    <= `EXE_RES_NOP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end
```

`*EXE_TNE*: begin // *tne*指令

```
wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_TNE_OP;
alusel_o    <= `EXE_RES_NOP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b1;
instvalid   <= `InstValid;
end
```

`*EXE_SYSCALL*: begin // *syscall*指令

```

        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_SYSCALL_OP;
        alusel_o    <= `EXE_RES_NOP;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        instvalid   <= `InstValid;
        excepttype_is_syscall<= `True_v;
    end
    default: begin
    end
endcase
end

.....
`EXE_REGIMM_INST: begin
    case (op4)
    .....

`EXE_TEQI: begin           // teqi指令
    wreg_o      <= `WriteDisable;
    aluop_o     <= `EXE_TEQI_OP;
    alusel_o    <= `EXE_RES_NOP;
    reg1_read_o <= 1'b1;

```

```

        reg2_read_o <= 1'b0;
                      imm             <= {{16{inst_i[15]}},
inst_i[15:0]};
                      instvalid    <= `InstValid;
end

`EXE_TGEI: begin // tgei指令

wreg_o      <= `WriteDisable;
aluop_o     <= `EXE_TGEI_OP;
alusel_o    <= `EXE_RES_NOP;
reg1_read_o <= 1'b1;
reg2_read_o <= 1'b0;
                      imm             <= {{16{inst_i[15]}},
inst_i[15:0]};
                      instvalid    <= `InstValid;
end

`EXE_TGEIU: begin // tgeiu指令

wreg_o      <= `WriteDisable;

```

```

        aluop_o      <= `EXE_TGEIU_OP;
        alusel_o     <= `EXE_RES_NOP;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        imm          <= {{16{inst_i[15]}}},
inst_i[15:0];
        instvalid   <= `InstValid;
    end

`EXE_TLTI: begin                                // tlti指令

        wreg_o      <= `WriteDisable;
        aluop_o      <= `EXE_TLTI_OP;
        alusel_o     <= `EXE_RES_NOP;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        imm          <= {{16{inst_i[15]}}},
inst_i[15:0];
        instvalid   <= `InstValid;
    end

`EXE_TLTIU: begin                               // tlтиu指令

```

```

        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_TLTIU_OP;
        alusel_o    <= `EXE_RES_NOP;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        imm          <= {{16{inst_i[15]}},
inst_i[15:0]};
        instinvalid <= `InstInvalid;
end

`EXE_TNEI: begin // tnei指令

        wreg_o      <= `WriteDisable;
        aluop_o     <= `EXE_TNEI_OP;
        alusel_o    <= `EXE_RES_NOP;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        imm          <= {{16{inst_i[15]}},
inst_i[15:0]};
        instinvalid <= `InstInvalid;
end

default: begin

```

```
        end  
    endcase  
end  
.....  
end  
  
default: begin  
end  
endcase //case op  
.....  
  
if(inst_i == `EXE_ERET) begin // eret指令  
  
wreg_o <= `WriteDisable;  
aluop_o <= `EXE_ERET_OP;  
alusel_o <= `EXE_RES_NOP;  
reg1_read_o <= 1'b0;  
reg2_read_o <= 1'b0;  
instvalid <= `InstValid;  
excepttype_is_eret<= `True_v;  
end .....
```

.....

变量excepttype_o收集译码阶段得到的异常信息，其第8bit表示是否是syscall指令引起的系统调用异常，第9bit表示是否是无效指令引起的异常，第12bit表示是否是返回指令eret。

译码工作主要是确定要写的目的寄存器、要读取的寄存器、要执行的运算三个方面信息。对其中几个典型指令的译码过程说明如下，其余指令的译码过程可以参考这几个典型指令。

(1) teq指令

- 要写的目的寄存器：teq指令不需要写通用寄存器，所以设置wreg_o为WriteDisable。
- 要读取的寄存器：teq指令需要读取地址为rs、rt的通用寄存器的值，所以设置reg1_read_o、reg2_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o是指令的21-25bit，正是teq指令中的rs（参考图11-4），默认通过Regfile模块读端口2读取的寄存器地址reg2_addr_o是指令的16-20bit，正是teq指令中的rt（参考图11-4）。所以最终译码阶段的输出reg1_o就是地址为rs的寄存器的值，reg2_o就是地址为rt的寄存器的值。
- 要执行的运算：因为teq指令不需要写通用寄存器，所以设置alusel_o为

EXE_RES_NOP。另外，设置aluop_o为EXE_TEQ_OP，表示运算子类型是teq。tlt、tge、tgeu、tltu、tne指令的译码过程可以参考teq指

令。

(2) teqi指令

- 要写的目的寄存器：teqi指令不需要写通用寄存器，所以设置wreg_o为WriteDisable。
- 要读取的寄存器：teqi指令需要读取地址为rs的通用寄存器的值，所以设置reg1_read_o为1。默认通过Regfile模块读端口1读取的寄存器地址reg1_addr_o是指令的第21-25bit，正是teqi指令中的rs（参考图11-4）。设置reg2_read_o为0，暗含使用立即数作为运算的操作数。imm就是指令中的立即数进行符号扩展后的值。所以最终译码阶段的输出reg1_o就是地址为rs的通用寄存器的值，reg2_o就是imm的值。
- 要执行的运算：因为teqi指令不需要写通用寄存器，所以设置alusel_o为

EXE_RES_NOP。另外，设置aluop_o为EXE_TEQI_OP，表示运算子类型是teqi。tlti、tgei、tgeiu、tltiu、tnei指令的译码过程可以参考teqi指令。

(3) syscall指令

- 要写的目的寄存器：syscall指令不需要写通用寄存器，所以设置wreg_o为WriteDisable。
- 要读取的寄存器：syscall指令不需要读取通用寄存器的值，所以设置reg1_read_o、reg2_read_o为0。
- 要执行的运算：因为syscall指令不需要写通用寄存器，所以设置alusel_o为EXE_RES_NOP。另外，设置aluop_o为

EXE_SYSCALL_OP，表示运算子类型是syscall。

- 此外，设置变量excepttype_is_syscall的值为True，表示当前异常类型是系统调用异常。

(4) eret指令

- 要写的目的寄存器：eret指令不需要写通用寄存器，所以设置wreg_o为WriteDisable。
- 要读取的寄存器：eret指令不需要读取通用寄存器的值，所以设置reg1_read_o、reg2_read_o为0。
- 要执行的运算：因为eret指令不需要写通用寄存器，所以设置alusel_o 为 EXE_RES_NOP 。另外，设置aluop_o 为 EXE_ERET_OP，表示运算子类型是eret。
- 此外，设置变量excepttype_is_eret的值为True，表示当前执行的是返回指令，也可以认为当前异常类型是返回异常。

2. 修改ID/EX模块

从图11-9可知，ID/EX模块需要增加部分接口，如表11-6所示。

表11-6 ID/EX模块增加接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	flush	1	输入	流水线清除信号
2	id_excepttype	32	输入	译码阶段收集到的异常信息
3	id_current_inst_addr	32	输入	译码阶段指令的地址
4	ex_excepttype	32	输出	译码阶段收集到的异常信息
5	ex_current_inst_addr	32	输出	执行阶段指令的地址

ID/EX模块的代码主要修改如下，完整代码可以参考本书附带光盘中Code\Chapter11目录下的id_ex.v文件。

```
module id_ex(
    .....
    input wire flush,
    // 新增的接口，从译码阶段传递过来的信号
    input wire[`RegBus] id_current_inst_address,
    input wire[31:0] id_excepttype,
    .....
    // 新增的接口，传递到执行阶段的信号
    output reg[`RegBus] ex_current_inst_address,
    output reg[31:0] ex_excepttype
);

always @ (posedge clk) begin
    if (rst == `RstEnable) begin //复位
        .....
        ex_excepttype <= `ZeroWord;
        ex_current_inst_address <= `ZeroWord;
    end else
    if(flush == 1'b1 ) begin //清除流水线
        ex_aluop <= `EXE_NOP_OP;
```

```

    ex_aluse1      <= `EXE_RES_NOP;
    ex_reg1        <= `ZeroWord;
    ex_reg2        <= `ZeroWord;
    ex_wd          <= `NOPRegAddr;
    ex_wreg         <= `WriteDisable;
    ex_excepttype   <= `ZeroWord;
    ex_link_address <= `ZeroWord;
    ex_inst         <= `ZeroWord;
    ex_is_in_delayslot <= `NotInDelaySlot;
    is_in_delayslot_o  <= `NotInDelaySlot;
    ex_current_inst_address <= `ZeroWord;

end else if(stall[2] == `Stop && stall[3] == `NoStop)
begin
    .....
    //译码阶段
暂停, 执行阶段没有暂停
    ex_excepttype <= `ZeroWord;
    ex_current_inst_address <= `ZeroWord;

end else if(stall[2] == `NoStop) begin //译码阶段没有
暂停
    .....

```

```
ex_excepttype <= id_excepttype;  
                                ex_current_inst_address <=  
id_current_inst_address;  
  
end  
end  
  
endmodule
```

上述代码主要修改的地方如下。

- 在有流水线清除事件时（即flush为1），将ID/EX模块中的所有寄存器设置为初始值。
- 在没有流水线清除事件（即flush为0），并且译码阶段没有暂停的情况下（即stall[2]为NoStop），将译码阶段得到的异常信息id_excepttype、指令地址id_current_inst_address传递到执行阶段。

11.6.3 修改执行阶段

1. 修改EX模块

EX模块接收从译码阶段传递过来的信息，同时，还要进一步判断是否有自陷异常，或者溢出异常发生。从图11-9可知，EX模块需要增

加部分接口，如表11-7所示。

表11-7 EX模块增加的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	excepttype_i	32	输入	译码阶段收集到的异常信息
2	current_inst_addr_i	32	输入	执行阶段指令的地址
3	excepttype_o	32	输出	译码阶段、执行阶段收集到的异常信息
4	current_inst_addr_o	32	输出	执行阶段指令的地址
5	is_in_delayslot_o	1	输出	执行阶段的指令是否是延迟槽指令

EX模块的代码主要修改如下，完整代码可以参考本书附带光盘中Code\Chapter11目录下的ex.v文件。

```
module ex(  
    . . . . .  
    // 新增的输入接口  
    input wire[31:0]           excepttype_i,  
    input wire[`RegBus]         current_inst_address_i,  
    . . . . .  
    // 新增的输出接口  
    output wire[31:0]          excepttype_o,  
    output wire                is_in_delayslot_o,  
    output wire[`RegBus]        current_inst_address_o,  
    . . . . .
```

```
);

.....



reg trapassert;           // 新定义变量，表示是否有自陷异常
reg ovassert;             // 新定义变量，表示是否有溢出异常

.....



// 执行阶段输出的异常信息就是译码阶段的异常信息加上自陷异常、溢出异常的信息，

// 其中第10bit表示是否有自陷异常，第11bit表示是否有溢出异常
assign excepttype_o = {excepttype_i[31:12],
ovassert, trapassert,

excepttype_i[9:8], 8'h00};

// is_in_delayslot_i表示当前指令是否在延迟槽中，在第8章已有详细介绍
assign is_in_delayslot_o = is_in_delayslot_i;

// 当前处于执行阶段指令的地址
assign current_inst_address_o = current_inst_address_i;
```

```

/*
**
***** 第一段：计算以下4个变量的值 *****
*****
**/


// (1) 如果是减法运算、有符号比较运算、有符号自陷指令，那么reg2_i_mux
等于

//      第二个操作数reg2_i的补码，否则reg2_i_mux就等于reg2_i

assign reg2_i_mux = ((aluop_i == `EXE_SUB_OP) ||
                      (aluop_i == `EXE_SUBU_OP) ||
                      (aluop_i == `EXE_SLT_OP) ||
                      (aluop_i == `EXE_TLT_OP) ||
                      (aluop_i == `EXE_TLTI_OP) ||
                      (aluop_i == `EXE_TGE_OP) ||
                      (aluop_i == `EXE_TGEI_OP)) ?
                      (~reg2_i)+1 : reg2_i;

// (2) 计算变量result_sum的值，有三种情况。
//      A. 如果是加法运算，此时reg2_i_mux就是reg2_i，所以result_sum
//          就是加法的结果
//      B. 如果是减法运算，此时reg2_i_mux等于reg2_i的补码，所以

```

```
//         result_sum就是减法运算的结果  
//      c. 如果是比较运算或有符号自陷指令，此时reg2_i_mux也等于reg2_i  
//          的补码，所以result_sum也是减法运算的结果，可以通过判断减法  
//          的结果是否小于零，进而判断第一个操作数reg1_i是否小于第二个操  
//          作数reg2_i
```

```
assign result_sum = reg1_i + reg2_i_mux;
```

```
// (3) 计算是否溢出，加法指令add、addi、减法sub指令执行的时候，  
//      需要判断是否溢出，满足以下两种情况之一时，有溢出  
//      A. reg1_i为正数，reg2_i_mux为正数，但是两者之和为负数  
//      B. reg1_i为负数，reg2_i_mux为负数，但是两者之和为正数
```

```
assign ov_sum = ((!reg1_i[31] && !reg2_i_mux[31]) &&  
result_sum[31])  
||((reg1_i[31] && reg2_i_mux[31]) &&  
(!result_sum[31]));
```

```
// (4) 计算操作数1是否小于操作数2，分两种情况
```

```

//      A. 当前指令为有符号比较指令或者有符号自陷异常指令的时候，此时又分
3种情况
//          A1. reg1_i为负数、reg2_i为正数，显然reg1_i小于reg2_i
//          A2. reg1_i为正数、reg2_i为正数，并且reg1_i减去reg2_i的值
小于0
//              (即result_sum为负)，此时也有reg1_i小于reg2_i
//          A3. reg1_i为负数、reg2_i为负数，并且reg1_i减去reg2_i的值
小于0
//              (即result_sum为负)，此时也有reg1_i小于reg2_i
//      B. 当前指令为无符号比较指令或者无符号自陷异常指令的时候，直接使
用比较运算符
//          比较reg1_i与reg2_i

assign reg1_lt_reg2 = ((aluop_i == `EXE_SLT_OP) ||

                      (aluop_i == `EXE_TLT_OP) ||

                      (aluop_i == `EXE_TLTI_OP) ||

                      (aluop_i == `EXE_TGE_OP) ||

                      (aluop_i == `EXE_TGEI_OP)) ?

                      ((reg1_i[31] && !reg2_i[31]) ||

                       (!reg1_i[31] && !reg2_i[31] &&

                        result_sum[31])) ||

                      (reg1_i[31] && reg2_i[31] &&

                        result_sum[31])))

: (reg1_i < reg2_i);

```

```
....  
  
/*********************************************  
**  
***** 第二段：判断是否发生自陷异常 *****  
*****  
**/  
  
//依据上面得到的比较结果，判断是否满足自陷指令的条件，从而给出变量  
trapassert的值  
  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        trapassert <= `TrapNotAssert;  
    end else begin  
  
        trapassert <= `TrapNotAssert;      //默认没有自陷异常  
  
        case (aluop_i)  
  
        // teg、teqi指令
```

```
`EXE_TEQ_OP, `EXE_TEQI_OP: begin
    if( reg1_i == reg2_i ) begin
        trapassert <= `TrapAssert;
    end
end
```

// *tge*、*tgei*、*tgeiu*、*tgeu*指令

```
`EXE_TGE_OP, `EXE_TGEI_OP, `EXE_TGEIU_OP,
`EXE_TGEU_OP:
begin
    if( ~reg1_lt_reg2 ) begin
        trapassert <= `TrapAssert;
    end
end
```

// *tlt*、*tlti*、*tltiu*、*tltu*指令

```
`EXE_TLT_OP, `EXE_TLTI_OP, `EXE_TLTIU_OP,  
`EXE_TLTU_OP:  
begin  
    if( reg1_lt_reg2 ) begin  
        trapassert <= `TrapAssert;  
    end  
end  
  
// tne、tnei指令  
  
`EXE_TNE_OP, `EXE_TNEI_OP: begin  
    if( reg1_i != reg2_i ) begin  
        trapassert <= `TrapAssert;  
    end  
end  
  
default: begin  
    trapassert <= `TrapNotAssert;  
end  
endcase  
end
```

```
end

. . .

//****************************************************************************
** 第三段：判断是否发生溢出异常 ****
** */

//依据指令类型以及ov_sum的值，判断是否发生溢出异常，从而给出变量
ovassert的值

always @ (*) begin
    if(((aluop_i == `EXE_ADD_OP) || (aluop_i == `EXE_ADDI_OP)
||

    (aluop_i == `EXE_SUB_OP)) && (ov_sum == 1'b1)) begin
        wreg_o    <= `WriteDisable;

    ovassert <= 1'b1;                                //发生了溢出异常

end else begin
    wreg_o    <= wreg_i;

    ovassert <= 1'b0;                                //没有发生溢出异常
```

```
end  
.....  
end  
endmodule
```

上述代码可以分为三段理解。

(1) 第一段：计算出如下几个变量的值。

- reg2_i_mux：如果是减法运算、有符号比较运算、有符号自陷指令，那么reg2_i_mux等于第二个操作数reg2_i的补码，否则reg2_i_mux就等于reg2_i。
- result_sum：第一个操作数reg1_i与第二个操作数reg2_i相加的结果，或者第一个操作数reg1_i与第二个操作数reg2_i相减的结果。
- ov_sum：指示加、减法是否溢出。
- reg1_lt_reg2：操作数1是否小于操作数2。

(2) 第二段：判断是否满足自陷异常的条件，从而确定变量trapassert的值。以tge指令为例，如果reg1_lt_reg2为0，即不是操作数1小于操作数2，那么一定是操作数1大于等于操作数2，此时满足tge指

令的条件，会发生自陷异常，从而设置变量trapassert为TrapAssert，表示自陷异常发生。

(3) 第三段：判断是否满足溢出异常的条件，从而确定变量ovassert的值。在add、addi、sub指令的执行过程中，如果发生溢出(ov_sum为1)，那么会引起溢出异常，设置变量ovassert为1，表示溢出异常发生。

在执行阶段收集的异常信息与译码阶段收集的异常信息一起通过接口excepttype_o传递到访存阶段，其中第10bit表示是否有自陷异常，第11bit表示是否有溢出异常，参考上面代码中对excepttype_o的赋值。

同时传递到访存阶段的还有指令地址、是否是延迟槽指令等信息，从图11-3可知，当异常发生时，这两个信息用来确定保存到EPC寄存器的值。

2. 修改EX/MEM模块

EX/MEM模块接收从EX模块送入的信号，将其传递到访存阶段，从图11-9可知，EX/MEM模块需要增加如表11-8所示的接口。

表11-8 EX/MEM模块新增接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	flush	1	输入	流水线清除信号
2	ex_excepttype	32	输入	译码、执行阶段收集到的异常信息
3	ex_current_inst_address	32	输入	执行阶段指令的地址
4	ex_is_in_delayslot	1	输入	执行阶段的指令是否是延迟槽指令
5	mem_excepttype	32	输出	译码、执行阶段收集到的异常信息
6	mem_current_inst_address	32	输出	访存阶段指令的地址
7	mem_is_in_delayslot	1	输出	访存阶段的指令是否是延迟槽指令

EX/MEM模块的代码主要修改如下，完整代码位于本书附带光盘中Code\Chapter11目录下的ex_mem.v文件中。

```
module ex_mem(  
    .....  
  
    input wire[31:0]           ex_excepttype,  
    input wire                ex_is_in_delayslot,  
    input wire[`RegBus]        ex_current_inst_address,  
    .....  
  
    output reg[31:0]           mem_excepttype,  
    output reg                 mem_is_in_delayslot,  
    output reg[`RegBus]        mem_current_inst_address,  
    .....  
);  
  
always @ (posedge clk) begin  
    if(rst == `RstEnable) begin          //复位  
        .....  
        mem_excepttype      <= `ZeroWord;  
        mem_is_in_delayslot <= `NotInDelaySlot;  
        mem_current_inst_address <= `ZeroWord;
```

```

    end else

if(flush == 1'b1 ) begin          //清除流水线
    mem_wd <= `NOPRegAddr;
    mem_wreg <= `WriteDisable;
    mem_wdata <= `ZeroWord;
    mem_hi <= `ZeroWord;
    mem_lo <= `ZeroWord;
    mem_whilo <= `WriteDisable;
    mem_aluop <= `EXE_NOP_OP;
    mem_mem_addr <= `ZeroWord;
    mem_reg2 <= `ZeroWord;
    mem_cp0_reg_we <= `WriteDisable;
    mem_cp0_reg_write_addr <= 5'b00000;
    mem_cp0_reg_data <= `ZeroWord;
    mem_excepttype <= `ZeroWord;
    mem_is_in_delayslot <= `NotInDelaySlot;
    mem_current_inst_address <= `ZeroWord;
    hilo_o <= {`ZeroWord, `ZeroWord};
    cnt_o <= 2'b00;

end else if(stall[3] == `Stop && stall[4] == `NoStop)
begin

```

```
      .....
      //执行阶段暂停, 访存阶段

没有暂停

    mem_excepttype          <= `ZeroWord;
    mem_is_in_delayslot     <= `NotInDelaySlot;
    mem_current_inst_address <= `ZeroWord;

end else

if(stall[3] == `NoStop) begin   //执行阶段没有暂停

      .....

mem_excepttype          <= ex_excepttype;
    mem_is_in_delayslot     <= ex_is_in_delayslot;
    mem_current_inst_address <= ex_current_inst_address;

end else begin

      .....

end

end

endmodule
```

上述代码主要修改的地方如下。

- 在有流水线清除事件时（即flush为1），将EX/MEM模块中的所有寄存器设置为初始值。
- 在没有流水线清除事件（即flush为0），并且执行阶段没有暂停的情况下（即stall[3]为NoStop），将执行、译码阶段收集到的异常信息ex_excepttype、指令地址ex_current_inst_address、是否是延迟槽指令ex_is_in_delayslot等信息传递到访存阶段。

11.6.4 修改访存阶段

1. 修改MEM模块

OpenMIPS处理器会在访存阶段的MEM模块综合所有的异常信息、CP0寄存器的值，最终判断是否有要处理的异常。参考图11-9可知，MEM模块要增加部分接口，新增接口的描述如表11-9所示。

表11-9 MEM模块增加的接口

序号	接 口 名	宽 度 (bit)	输入/ 输出	作 用
1	excepttype_i	32	输入	译码、执行阶段收集到的异常信息
2	current_inst_address_i	32	输入	访存阶段指令的地址
3	is_in_delayslot_i	1	输入	访存阶段的指令是否是延迟槽指令
4	cp0_status_i	32	输入	CP0 中 Status 寄存器的值
5	cp0_cause_i	32	输入	CP0 中 Cause 寄存器的值
6	cp0_epc_i	32	输入	CP0 中 EPC 寄存器的值
7	wb_cp0_reg_we	1	输入	回写阶段的指令是否要写 CP0 中的寄存器
8	wb_cp0_reg_write_address	5	输入	回写阶段的指令要写的 CP0 中寄存器的地址
9	wb_cp0_reg_data	32	输入	回写阶段的指令要写入 CP0 中寄存器的值
10	excepttype_o	32	输出	最终的异常类型
11	current_inst_address_o	32	输出	访存阶段指令的地址
12	is_in_delayslot_o	1	输出	访存阶段的指令是否是延迟槽指令
13	cp0_epc_o	32	输出	CP0 中 EPC 寄存器的最新值

MEM模块的代码主要修改如下，完整代码可以参考本书光盘中 Code\Chapter11 目录下的mem.v文件。

```
module mem(
    . . . .
    // 新增加的接口，来自执行阶段
    input wire[31:0]           excepttype_i,
    input wire                is_in_delayslot_i,
    input wire[`RegBus]         current_inst_address_i,
    // 新增加的接口，来自CP0模块
    input wire[`RegBus]         cp0_status_i,
    input wire[`RegBus]         cp0_cause_i,
    input wire[`RegBus]         cp0_epc_i,
```



```
// is_in_delayslot_o表示访存阶段的指令是否是延迟槽指令

assign is_in_delayslot_o = is_in_delayslot_i;

// current_inst_address_o是访存阶段指令的地址

assign current_inst_address_o = current_inst_address_i;

.....
// *****
** 第一段：得到CP0中寄存器的最新值 *****
** */

// 得到CP0中Status寄存器的最新值，步骤如下：
// 判断当前处于回写阶段的指令是否要写CP0中Status寄存器，如果要写，那么
```

要写

```
// 入的值就是Status寄存器的最新值，反之，从CP0模块通过cp0_status_i  
接口
```

```
// 传入的数据就是Status寄存器的最新值  
  
always @ (*) begin  
  
    if(rst == `RstEnable) begin  
  
        cp0_status <= `ZeroWord;  
  
    end else if((wb_cp0_reg_we == `WriteEnable) &&  
(wb_cp0_reg_write_addr == `CP0_REG_STATUS ))begin  
  
        cp0_status <= wb_cp0_reg_data;  
  
    end else begin  
  
        cp0_status <= cp0_status_i;  
  
    end  
  
end
```

```
// 得到CP0中EPC寄存器的最新值，步骤如下：
```

```
// 判断当前处于回写阶段的指令是否要写CP0中EPC寄存器，如果要写，那么要  
写入
```

```
// 的值就是EPC寄存器的最新值，反之，从CP0模块通过cp0_epc_i接口传入的  
数
```

```
// 据就是EPC寄存器的最新值
```

```
always @ (*) begin  
  
    if(rst == `RstEnable) begin  
  
        cp0_epc <= `ZeroWord;  
  
    end else if((wb_cp0_reg_we == `WriteEnable) &&  
                (wb_cp0_reg_write_addr == `CP0_REG_EPC))begin  
  
        cp0_epc <= wb_cp0_reg_data;  
  
    end  
  
end
```

```

    cp0_epc <= wb_cp0_reg_data;
end else begin
    cp0_epc <= cp0_epc_i;
end
end

// 将EPC寄存器的最新值通过接口cp0_epc_o输出

assign cp0_epc_o = cp0_epc;

// 得到CP0中Cause寄存器的最新值，步骤如下：
// 判断当前处于回写阶段的指令是否要写CP0中Cause寄存器，如果要写，那么
要写入
// 的值就是Cause寄存器的最新值，不过注意一点：Cause寄存器只有几个字段
是可写
// 的。反之，从CP0模块通过cp0_cause_i接口传入的数据就是Cause寄存器的
最新
// 值
always @ (*) begin
    if(rst == `RstEnable) begin
        cp0_cause <= `ZeroWord;
    end else if((wb_cp0_reg_we == `WriteEnable) &&
                (wb_cp0_reg_write_addr == `CP0_REG_CAUSE)

```

```

))begin
    cp0_cause[9:8] <= wb_cp0_reg_data[9:8]; // IP[1:0]字段是
可写的
    cp0_cause[22] <= wb_cp0_reg_data[22]; // WP字段是可写的
    cp0_cause[23] <= wb_cp0_reg_data[23]; // IV字段是可写的
end else begin
    cp0_cause      <= cp0_cause_i;
end
end

```

```

/*****
**
*****          第二段：给出最终的异常类型          *****
*****
*/

```

```

always @ (*) begin
    if(rst == `RstEnable) begin
        excepttype_o <= `ZeroWord;
    end else begin
        excepttype_o <= `ZeroWord;
        if(current_inst_address_i != `ZeroWord) begin
            if(((cp0_cause[15:8] & (cp0_status[15:8])) != 8'h00)
&&
                (cp0_status[1] == 1'b0) &&
                (cp0_status[0] == 1'b1)) begin
            excepttype_o <= 32'h00000001;
        end
    end
end

```

```
//interrupt

    end else if(excepttype_i[8] == 1'b1) begin
        excepttype_o <= 32'h00000008;

//syscall

    end else if(excepttype_i[9] == 1'b1) begin
        excepttype_o <= 32'h0000000a;

//inst_invalid

    end else if(excepttype_i[10] == 1'b1) begin
        excepttype_o <= 32'h0000000d;

//trap

    end else if(excepttype_i[11] == 1'b1) begin
```

```
        excepttype_o <= 32'h0000000c;  
//ov  
  
end else if(excepttype_i[12] == 1'b1) begin  
    excepttype_o <= 32'h0000000e;  
//eret  
  
end  
end  
end  
end  
  
*****  
**  
***** 第三段：给出对数据存储器的写操作  
*****  
*****  
**/  
  
// mem_we_o输出到数据存储器，表示是否是对数据存储器的写操作，  
// 如果发生了异常，那么需要取消对数据存储器的写操作
```

```
assign mem_we_o = mem_we & (~(|excepttype_o));  
  
endmodule
```

上述代码可以分为三段理解。

(1) 第一段：从CP0模块传入的Status、EPC、Cause等寄存器的值并不一定是最newValue，因为处于回写阶段的指令可能要写这些寄存器，此处又是一种数据相关的情况，在之前已多次遇到，解决方法都是一样的——将数据前推。这里就是将回写阶段要写的CP0寄存器的信息前推到访存阶段，在访存阶段进行判断，从而得到Status、EPC、Cause寄存器的最新值。以Status寄存器为例，首先判断当前处于回写阶段的指令是否要写CP0中Status寄存器，如果要写，那么要写入的值（即wb_cp0_reg_data）就是Status寄存器的最新值，反之，从CP0模块传入的数据（即cp0_status_i）就是Status寄存器的最新值。

(2) 第二段：依据CP0中寄存器的值，以及译码、执行阶段收集到的异常类型，得到最终的异常类型。首先判断当前处于访存阶段指令的地址是否为0，如果为0，那么表示处理器处于复位状态，或者刚刚发生异常，正在清除流水线（flush为1），或者流水线处于暂停状态，在这三种情况下都不处理异常。

如果当前处于访存阶段指令的地址不为0，那么可以进一步判断有没有异常、是何种异常，从而给输出变量excepttype_o赋值。各种异常的判断依据列举如下，判断过程如图11-11所示。

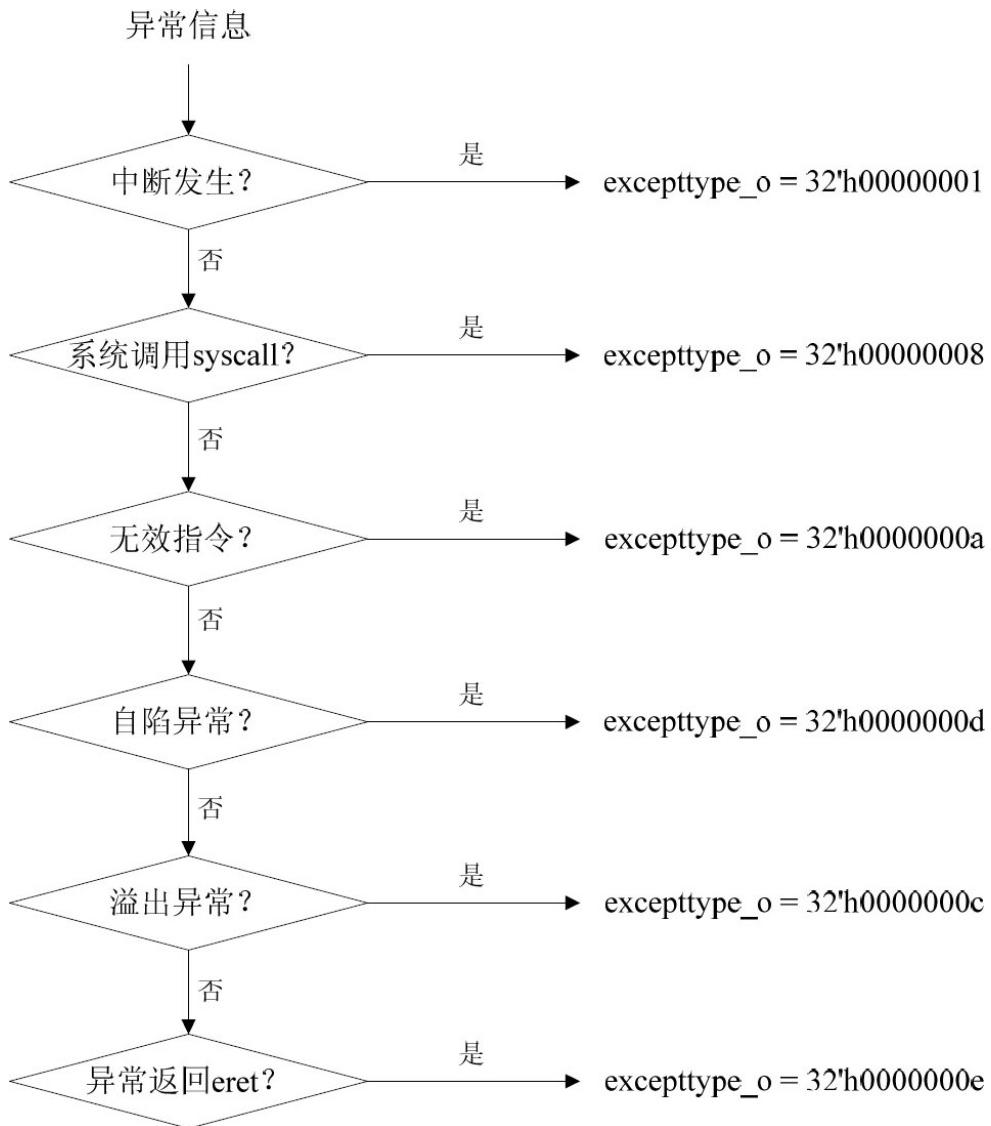


图11-11 异常类型的判断过程

- 发生中断的依据是： Cause寄存器的IP字段不为0，且Status寄存器中相应的中断掩码字段IM也不为0，另外，Status寄存器的EXL字段为0，表示不处于异常处理过程中，Status寄存器的IE字段为1，表示中断使能。
- 发生系统调用异常的依据是：输入的异常类型excepttype_i的第8bit为1，具体原因可以参考11.6.2节译码阶段中syscall指令的译码过程。

- 发生无效指令异常的依据是：输入的异常类型excepttype_i的第9bit为1，具体原因可以参考11.6.2节译码阶段。
- 发生自陷异常的依据是：输入的异常类型excepttype_i的第10bit为1，具体原因可以参考11.6.3节执行阶段对自陷指令的处理过程。
- 发生溢出异常的依据是：输入的异常类型excepttype_i的第11bit为1，具体原因可以参考11.6.3节执行阶段对溢出情况的处理过程。
- 确定是异常返回指令eret的依据是：输入的异常类型excepttype_i的第12bit为1，具体原因可以参考11.6.2节译码阶段中eret指令的译码过程。

(3) 第三段：OpenMIPS处理器要实现精确异常，也就是发生异常时，引起异常的指令及其后面已经进入流水线的指令都会失效。如果引起异常的指令是存储指令，那么要使其失效，就要停止修改数据存储器，所以在这里修改mem_we_o的赋值，如果发生异常（即变量excepttype_o不为0），那么设置mem_we_o为0，从而不会修改数据存储器。

2. 修改MEM/WB模块

MEM/WB模块接收来自MEM模块的信号，将其传递到回写阶段，从图11-9可知，MEM/WB模块需要增加如表11-10所示的接口。

表11-10 MEM/WB模块新增接口的描述

序号	接口名	宽度(bit)	输入/输出	作用
1	flush	1	输入	流水线清除信号

MEM/WB模块的代码主要修改如下，完整代码位于本书附带光盘中Code\Chapter11目录下的mem_wb.v文件内。

```
module mem_wb(  
    .....  
    input wire      flush,  
    .....  
);  
  
always @ (posedge clk) begin  
    if(rst == `RstEnable) begin  
        .....  
    end else if(flush == 1'b1 ) begin //清除流水线  
        wb_wd          <= `NOPRegAddr;  
        wb_wreg         <= `WriteDisable;  
        wb_wdata         <= `ZeroWord;  
        wb_hi          <= `ZeroWord;  
        wb_lo          <= `ZeroWord;
```

```

wb_whilo           <= `WriteDisable;
wb_LLbit_we       <= 1'b0;
wb_LLbit_value    <= 1'b0;
wb_cp0_reg_we     <= `WriteDisable;
wb_cp0_reg_write_addr <= 5'b00000;
wb_cp0_reg_data   <= `ZeroWord;

.....
endmodule

```

如果发生了异常，导致flush信号为1，那么会放弃对通用寄存器、HI、LO、LLbit以及CP0中寄存器的修改，目的是不对处理器产生任何影响。

11.6.5 修改协处理器CP0

参考图11-9可知，访存阶段的一些输出会送入CP0模块，用来确定CP0中部分寄存器的值。CP0模块要增加的接口如表11-11所示。

表11-11 CP0模块新增接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	excepttype_o	32	输入	最终的异常类型
2	current_inst_address_o	32	输入	发生异常的指令地址
3	is_in_delayslot_o	1	输入	发生异常的指令是否是延迟槽指令

CP0模块的代码主要修改如下，完整代码请参考本书附带光盘中Code\Chapter11目录下的cp0_reg.v文件。

```
module cp0_reg(
    .....
    input wire[31:0]           excepttype_i,
    input wire[`RegBus]        current_inst_addr_i,
    input wire                is_in_delayslot_i,
    .....
);

always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        .....
    end else begin
        .....
    case (excepttype_i)
        32'h00000001: begin          //外部中断
            if(is_in_delayslot_i == `InDelaySlot ) begin
                epc_o      <= current_inst_addr_i - 4 ;
            end
        endcase
    end
end
```

```

        cause_o[31] <= 1'b1;           // Cause寄存器的BD字
段

    end else begin

        epc_o          <= current_inst_addr_i;
        cause_o[31] <= 1'b0;

    end

        status_o[1]      <= 1'b1;           // Status寄存器的
EXL字段

        cause_o[6:2]     <= 5'b00000;           // Cause寄存器的
ExcCode字段

    end

32'h00000008:      begin      //系统调用异常syscall

if(status_o[1] == 1'b0) begin
    if(is_in_delayslot_i == `InDelaySlot ) begin
        epc_o          <= current_inst_addr_i - 4 ;
        cause_o[31] <= 1'b1;

    end else begin
        epc_o          <= current_inst_addr_i;
        cause_o[31] <= 1'b0;

    end
end

```

```
    status_o[1]  <= 1'b1;  
    cause_o[6:2] <= 5'b01000;  
end
```

32'h0000000a: begin //无效指令异常

```
if(status_o[1] == 1'b0) begin  
    if(is_in_delayslot_i == `InDelaySlot ) begin  
        epc_o      <= current_inst_addr_i - 4 ;  
        cause_o[31] <= 1'b1;  
    end else begin  
        epc_o      <= current_inst_addr_i;  
        cause_o[31] <= 1'b0;  
    end  
end  
status_o[1]  <= 1'b1;  
cause_o[6:2] <= 5'b01010;  
end
```

32'h0000000d: begin //自陷异常

```
if(status_o[1] == 1'b0) begin
    if(is_in_delayslot_i == `InDelaySlot ) begin
        epc_o          <= current_inst_addr_i - 4 ;
        cause_o[31] <= 1'b1;
    end else begin
        epc_o          <= current_inst_addr_i;
        cause_o[31] <= 1'b0;
    end
end
status_o[1]  <= 1'b1;
cause_o[6:2] <= 5'b01101;
end
```

32'h0000000c: begin //溢出异常

```
if(status_o[1] == 1'b0) begin
    if(is_in_delayslot_i == `InDelaySlot ) begin
        epc_o          <= current_inst_addr_i - 4 ;
        cause_o[31] <= 1'b1;
    end else begin
        epc_o          <= current_inst_addr_i;
```

```
        cause_o[31] <= 1'b0;
    end
    status_o[1]  <= 1'b1;
    cause_o[6:2] <= 5'b01100;
end

32'h0000000e: begin //异常返回指令eret
    status_o[1] <= 1'b0;
end

default: begin
end
endcase
end
end
```

上述代码主要是依据访存阶段给出的最终异常类型，进而修改CP0中相应寄存器的值。不同异常类型修改的值也不同，依次介绍如下。

1. 中断

依据发生异常的指令是否位于延迟槽中，设置EPC寄存器的值，以及Status寄存器的BD字段，如果位于延迟槽中，那么设置EPC寄存器为上一条指令的地址，Status寄存器的BD字段为1，反之，设置EPC寄存器为发生异常指令的地址，Status寄存器的BD字段为0。另外，设置Status寄存器的EXL字段为1，表示处于异常级，中断禁止。最后，设置Cause寄存器的ExcCode字段为5'b00000，表示异常原因是中断，参考第10章的表10-7。

2. 系统调用异常

分两种情况。

(1) 如果Status寄存器的EXL字段为0，那么依据发生异常的指令是否位于延迟槽中，设置EPC寄存器的值，以及Status寄存器的BD字段。如果位于延迟槽中，那么设置EPC寄存器为上一条指令的地址，Status寄存器的BD字段为1，反之，设置EPC寄存器为发生异常指令的地址，Status寄存器的BD字段为0。然后，设置Status寄存器的EXL字段为1，表示处于异常级，中断禁止。最后，设置Cause寄存器的ExcCode字段为5'b01000，表示异常原因是系统调用指令syscall，参考第10章的表10-7。

(2) 如果Status寄存器的EXL字段为1，表示当前已经处于异常级了，又发生了新的异常，那么只需要将异常原因保存到Cause寄存器的

ExcCode字段，此处设置为5'b01000，表示异常原因是系统调用指令syscall。

3. 无效指令异常

与系统调用异常的处理过程类似，只是设置Status寄存器的ExcCode字段为5'b01010，表示异常原因是无效指令，参考第10章的表10-7。

4. 自陷异常

与系统调用异常的处理过程类似，只是设置Status寄存器的ExcCode字段为5'b01101，表示异常原因是自陷，参考第10章的表10-7。

5. 溢出异常

与系统调用异常的处理过程类似，只是设置Status寄存器的ExcCode字段为5'b01100，表示异常原因是溢出，参考第10章的表10-7。

6. 异常返回指令`eret`

清除Status寄存器的IE字段，表示中断允许。

上述6种异常的处理过程是与11.3节异常处理过程一致的。

11.6.6 修改控制模块CTRL

CTRL模块会依据异常类型，给出新的取指地址（即：异常处理例程入口地址），同时决定是否要清除流水线。其新增接口如表11-12所示，其中的输入信号excepttype_i、cp0_epc_i都来自MEM模块，读者可以参考图11-9。

表11-12 CTRL模块新增接口的描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	cp0_epc_i	32	输入	EPC 寄存器的最新值
2	excepttype_i	32	输入	最终的异常类型
3	new_pc	32	输出	异常处理入口地址
4	flush	1	输出	是否清除流水线

修改 CTRL 模块的代码如下，源文件是本书附带光盘 Code\Chapter11目录下的ctrl.v文件。

```
module ctrl(  
  
    input wire          rst,  
    input wire          stallreq_from_id,  
    input wire          stallreq_from_ex,  
  
    // 新增输入信号，来自MEM模块  
    input wire[31:0]      excepttype_i,  
    input wire[`RegBus]    cp0_epc_i,  
  
    // 新增输出信号  
    output reg[`RegBus]   new_pc,  
    output reg             flush,
```

```
    output reg[5:0]          stall
);

always @ (*) begin
    if(rst == `RstEnable) begin
        stall  <= 6'b000000;
        flush  <= 1'b0;
        new_pc <= `ZeroWord;
    end else if(excepttype_i != `ZeroWord) begin
// 不为0，表示发生异常

        flush  <= 1'b1;
        stall  <= 6'b000000;
        case (excepttype_i)
            32'h00000001: begin          // 中断
                new_pc <= 32'h00000020;
            end
        endcase
    end
end
```

```
32'h00000008: begin // 系统调用异常syscall
    new_pc <= 32'h00000040;
end

32'h0000000a: begin // 无效指令异常
    new_pc <= 32'h00000040;
end

32'h0000000d: begin // 自陷异常
    new_pc <= 32'h00000040;
end

32'h0000000c: begin // 溢出异常
```

```
    new_pc <= 32'h00000040;  
  end  
  
32'h0000000e: begin // 异常返回指令eret  
  
    new_pc <= cp0_epc_i;  
  end  
  default : begin  
  end  
endcase  
end else if(stallreq_from_ex == `Stop) begin  
  stall <= 6'b001111;  
  flush <= 1'b0;  
end else if(stallreq_from_id == `Stop) begin  
  stall <= 6'b000111;  
  flush <= 1'b0;  
end else begin  
  stall <= 6'b000000;  
  flush <= 1'b0;  
  new_pc <= `ZeroWord;  
end //if  
end //always
```

```
endmodule
```

当发生异常时（excepttype_i不为0），依据异常类型，设置输出信号new_pc为异常处理例程入口地址。同时设置输出信号flush为1。

对于指令eret而言，因为需要返回到异常发生前的状态继续执行，所以设置输出信号new_pc为EPC寄存器的值（即cp0_epc_i）。

参考图11-9可知，new_pc传递到PC模块，从11.6.1节对PC模块的修改可知，new_pc会作为新的取指地址。

读者如果耐心地读到这里，应该就知道如何自定义异常处理例程入口地址了。

11.6.7 修改OpenMIPS

最后，要修改OpenMIPS模块，在其中将上面修改的模块按照图11-9所示的关系连接起来。注意一点，在前几章已实现了LLbit模块的flush接口、DIV模块的annul_i接口，但是当时对这两个接口都没有连接有效信号，现在可以连接了，就连接到CTRL模块的输出接口flush。具体代码不在书中给出，读者可以参考本书附带光盘中Code\Chapter11目录下的openmips.v文件。

11.7 再次修改最小SOPC

在第9章，为了测试加载存储指令是否实现正确而修改了最小SOPC，本节将再次修改最小SOPC，将时钟中断输出作为一个中断信号输入，这样就可以处理时钟中断了，从而验证异常相关指令是否实现正确。如图11-12所示。

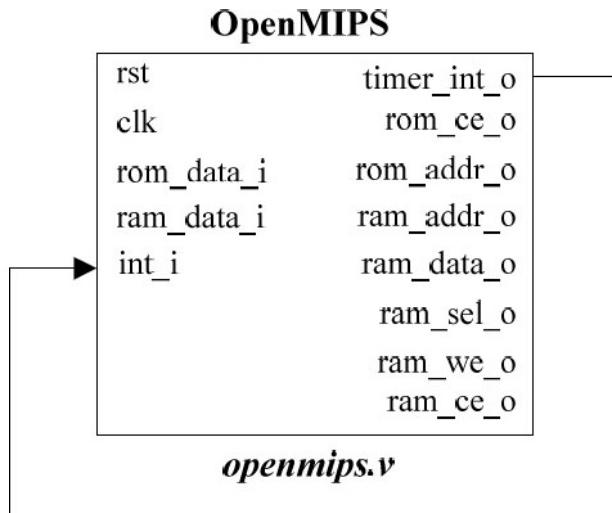


图11-12 再次修改最小SOPC以实现对时钟中断的处理

其中，输入接口int_i的宽度为6，时钟中断输出接口timer_int_o连接到int_i的最低位，修改openmips_min_sopc的代码如下，以实现图11-12所示的连接，完整代码请参考本书附带光盘中Code\Chapter11目录下的openmips_min_sopc.v文件。

```
module openmips_min_sopc(  
    input wire    clk,  
    input wire    rst  
);  
    .....  
endmodule
```

```
wire[3:0] mem_sel_i;  
wire[5:0] int;  
wire      timer_int;  
  
assign int = {5'b00000, timer_int};      // 时钟中断输出作为一个中断  
输入  
  
openmips openmips0(  
    .clk(clk),  
    .rst(rst),  
    .rom_addr_o(inst_addr),  
    .rom_data_i(inst),  
    .int_i(int),           // 中断输入
```

```
.ram_we_o(mem_we_i),  
.ram_addr_o(mem_addr_i),  
.ram_sel_o(mem_sel_i),  
.ram_data_o(mem_data_i),  
.ram_data_i(mem_data_o),  
  
.timer_int_o(timer_int)      //时钟中断输出  
  
);  
  
.....  
  
endmodule
```

11.8 测试程序

11.8.1 测试程序1——测试系统调用 异常

对系统调用异常的测试程序如下所示，源文件是本书附带光盘中 Code\Chapter11\AsmTest\test1 目录下的 inst_rom.S 文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:
    # 因为低地址有异常处理例程，所以处理器启动后，就立即转移到0x100处
    ori $1,$0,0x100      # (1) 设置寄存器$1 = 0x100
    jr  $1                 # 转移到地址0x100处
nop

    # 系统调用异常处理例程
    .org 0x40
    ori  $1,$0,0x8000      # (3) 设置寄存器$1 = 0x00008000
    ori  $1,$0,0x9000      # (4) 设置寄存器$1 = 0x00009000
    mfco $1,$14,0x0        # (5) 获取EPC寄存器的值保存到寄存器$1, $1 =
0x0000010c,
    addi $1,$1,0x4         # (6) 寄存器$1加4, $1 = 0x00000110
    mtc0 $1,$14,0x0        # 将EPC+4的结果保存回EPC寄存器
    eret
nop

    # 主程序，在其中调用syscall指令，从而引起系统调用异常
    .org 0x100
```

```
ori $1,$0,0x1000      # (2) 设置寄存器$1 = 0x1000

sw  $1, 0x0100($0)    # 将寄存器$1的值存储到内存0x100处,
# [0x100] = 0x00001000

mthi $1                # 将寄存器$1的值复制到寄存器HI, HI =
0x00001000

syscall                # 调用syscall指令, 引起系统调用异常

lw   $1, 0x0100($0)    # (7) 从内存0x100处加载数据, 保存到寄存器$1
# 所以, 最后$1 = 0x00001000

mfhi $2                # 将寄存器HI的值赋给寄存器$2, $2 =
0x00001000

loop:
j loop
nop
```

因为低地址有异常处理例程，所以处理器启动后，就立即转移到0x100处，随后调用syscall指令，引起系统调用异常，转移到0x40处进行异常处理。在异常处理例程中，读出EPC寄存器的值，将其加4，再保存回EPC寄存器，然后使用指令eret返回，继续执行。

注意：之所以要将EPC寄存器加4，是因为EPC寄存器中保存的是syscall指令的地址，如果不加4，那么eret指令会使得程序又回到syscall

指令处，从而又引发系统调用异常，永不停止，所以将EPC寄存器加4，这样当使用eret指令返回时，会返回到syscall指令的下一条指令处，从而避免了上述问题。

观察寄存器\$1的变化可以知道OpenMIPS是否正确实现了对系统调用异常的处理。\$1寄存器的变化顺序在程序的注释中已经注明。ModelSim仿真结果如图11-13所示，从中可知，OpenMIPS正确实现了对系统调用异常的处理。

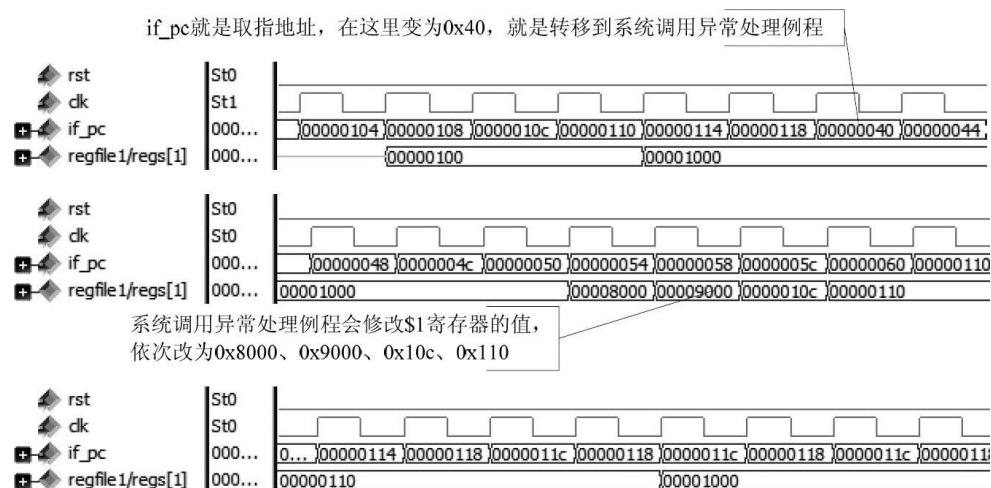


图11-13 测试程序1的ModelSim仿真结果

11.8.2 测试程序2——测试自陷异常

对自陷异常的测试程序如下所示，源文件是本书光盘中Code\Chapter11\AsmTest\test2目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder
```

```
.set nomacro
.global _start

_start:
    # 因为低地址有异常处理例程，所以处理器启动后，就立即转移到0x100处
    ori $1,$0,0x100    # 设置寄存器$1 = 0x100
    jr  $1                # 转移到地址0x100处
    nop

    # 自陷异常的处理例程，在其中设置寄存器$1的值
    .org 0x40
    ori $1,$0,0xf0f0      # 设置寄存器$1 = 0x0000f0f0
    ori $1,$0,0xffff       # 设置寄存器$1 = 0x0000ffff
    ori $1,$0,0x0f0f       # 设置寄存器$1 = 0x00000f0f

    mfcc0 $4,$14,0x0        # 获取EPC寄存器的值，保存到寄存器$4
    addi $4,$4,0x4          # 将寄存器$4加4
    mtc0 $4,$14,0x0        # 将寄存器$4的值赋给EPC，
    # 上面三条指令实际就是将EPC寄存器的值加4

    eret                  # 异常返回
    nop

    # 主程序，在其中会调用多个自陷指令
    .org 0x100
    ori $1,$0,0x1000      # $1 = 0x00001000
    ori $2,$0,0x1000      # $2 = 0x00001000
    teq $1,$2              # 此时$1等于$2，所以发生自陷异常
```

```
ori $1,$0,0x2000      # $1 = 0x00002000
tne $1,$2              # 此时$1不等于$2，所以发生自陷异常

ori $1,$0,0x3000      # $1 = 0x00003000
teqi $1,0x3000        # 此时$1等于0x3000，所以发生自陷异常

ori $1,$0,0x4000      # $1 = 0x00004000
tnei $1,0x2000        # 此时$1不等于0x2000，所以发生自陷异常

ori $1,$0,0x5000      # $1 = 0x00005000
tge $1,$2              # 此时$1大于$2，所以发生自陷异常

ori $1,$0,0x6000      # $1 = 0x00006000
tgei $1,0x4000        # 此时$1大于0x4000，所以发生自陷异常

ori $1,$0,0x7000      # $1 = 0x00007000
tgeiu $1,0x7000       # 此时$1等于0x7000，所以发生自陷异常

ori $1,$0,0x8000      # $1 = 0x00008000
tgeu $1,$2              # 此时$1大于$2，所以发生自陷异常

ori $1,$0,0x9000      # $1 = 0x00009000
tlr $1,$2              # 此时$1不小于$2，所以
```

不发生自陷异常

```
ori $1,$0,0xa000      # $1 = 0x0000a000
tlti $1,0x9000        # 此时$1不小于0xfffff9000, 所以
```

不发生自陷异常

```
ori $1,$0,0xb000      # $1 = 0x0000b000
tlти $1,0xb000        # 此时$1小于0xfffffb000, 所以发生自陷异常
```

```
ori $1,$0,0xc000      # $1 = 0x0000c000
tltu $2,$1             # 此时$2小于$1, 所以发生自陷异常
```

```
ori $1,$0,0xd000      # 最后, 设置寄存器$1 = 0x0000d000
```

```
_loop:
    j _loop
    nop
```

程序一开始就转移到地址0x100处的主程序中，在主程序会调用各种自陷指令，如果满足自陷指令的条件，那么会转移到自陷异常处理例程。注意：在自陷异常处理例程中，要将EPC寄存器的值加4，然后再调用指令eret返回，具体原因在测试程序1中已说明。

观察寄存器\$1的变化可以判断自陷指令、自陷异常处理是否实现正确，寄存器\$1的变化顺序在程序的注释中已经注明。ModelSim仿真结果如图11-14、图11-15所示，从中可知，OpenMIPS正确实现了自陷指令、自陷异常处理。

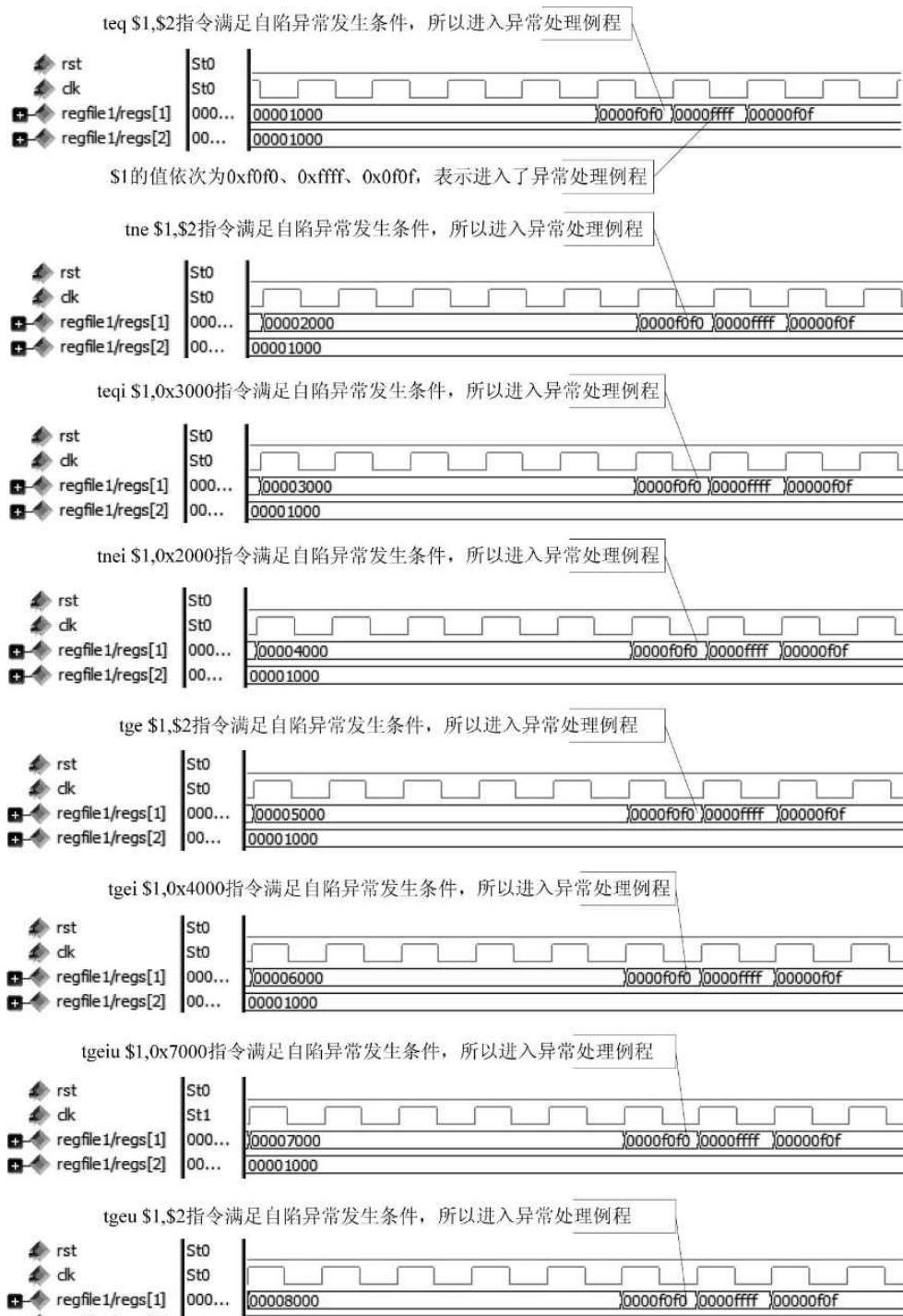


图11-14 测试程序2的ModelSim仿真结果

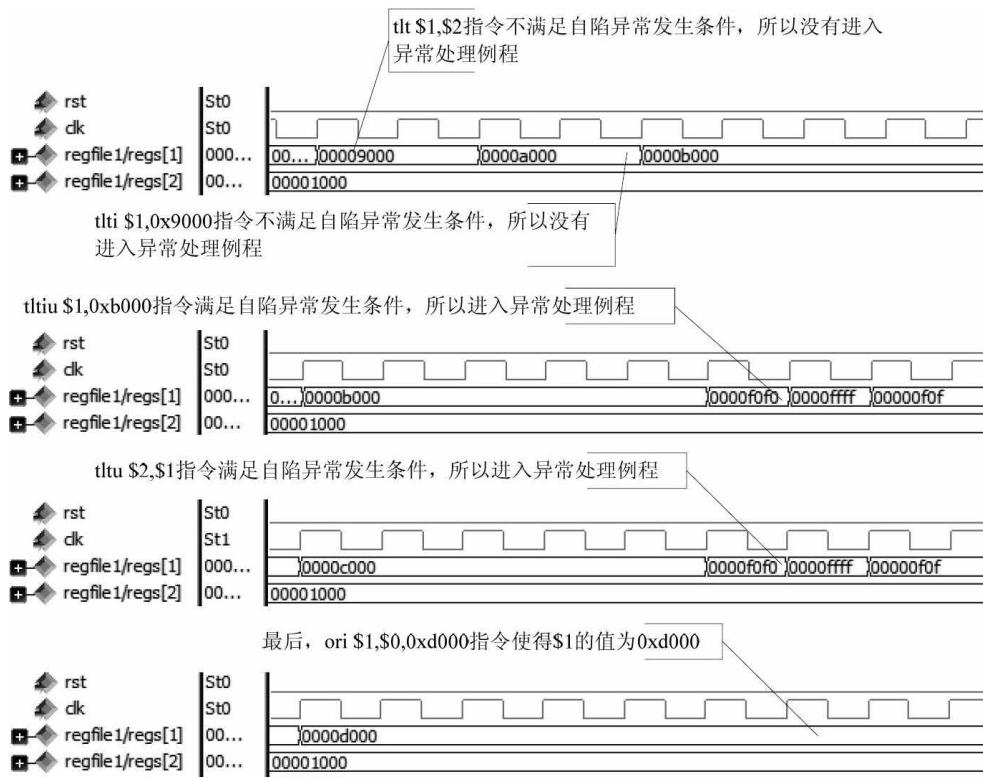


图11-15 测试程序2的ModelSim仿真结果（续）

11.8.3 测试程序3——测试时钟中断

时钟中断的测试程序如下所示，源文件是本书光盘中 Code\Chapter11\AsmTest\test3 目录下的 inst_rom.S 文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:
```

```
# 因为低地址有异常处理例程，所以处理器启动后，就立即转移到0x100处
ori $1,$0,0x100      # $1 = 0x100
jr  $1
nop

# 中断处理例程，在其中将$2寄存器的值加1，这样便于观察时钟中断是否发生。
# 另外，增加Compare寄存器的值，以清除时钟中断声明，同时设置下一次时钟
# 中断发生的时间

.org 0x20
addi $2,$2,0x1      # $2寄存器的值加1
mfco $1,$11,0x0      # 读取Compare寄存器的值
addi $1,$1,100       # 增加100
mtco $1,$11,0x0      # 再保存回Compare寄存器
eret
nop

# 主程序，在其中初始化Compare寄存器，并且使能时钟中断
.org 0x100
ori $2,$0,0x0
ori $1,$0,100        #
mtco $1,$11,0x0      # 初始化Compare寄存器的值为100

lui  $1,0x1000
ori  $1,$1,0x401      #
mtco $1,$12,0x0      # 设置Status寄存器的值为0x10000401，表示使
能时钟中断
```

```
_loop:  
    j _loop  
    nop
```

同前面的测试程序一样，程序一开始就转移到0x100处的主程序中，在其中初始化Compare寄存器的值，设置为100。然后设置Status寄存器的值为0x10000401，也就是设置其中的IE字段为1，表示中断使能，同时设置时钟中断对应的掩码IM[3]为1，从而允许时钟中断。然后进入一个循环等待中。

当Count寄存器的值等于Compare寄存器的值时，会发生时钟中断，导致进入中断处理例程，在其中将\$2寄存器的值加1，主要是方便观察仿真结果。然后将Compare寄存器的值加100，这样做一方面会清除当前的时钟中断声明，另一方面也设置了下一次时钟中断的发生时间。当Count寄存器的值再次等于Compare寄存器的值时，又会发生时钟中断。

观察\$2寄存器的变化可以知道OpenMIPS对时钟中断的处理是否实现正确。实现正确时，\$2寄存器的值应该不断累加。ModelSim仿真结果如图11-16所示，从中可知，OpenMIPS正确实现了对时钟中断的处理。

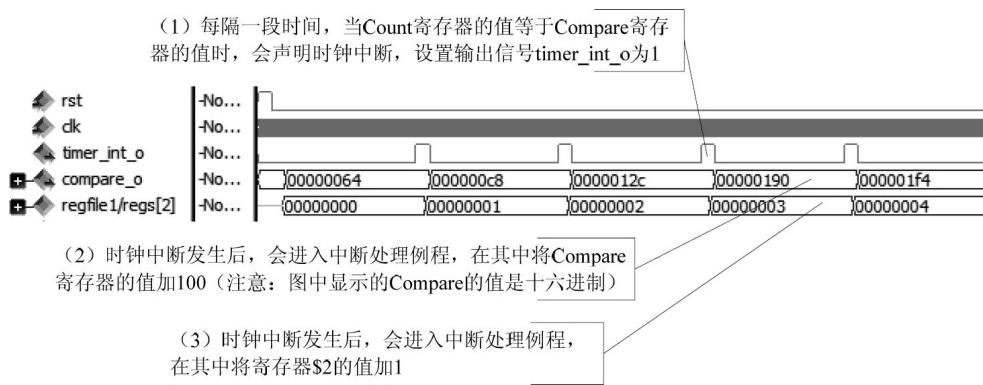


图11-16 测试程序3的ModelSim仿真结果

11.9 教学版OpenMIPS处理器实现小结

好了，我们的教学版OpenMIPS处理器已经完全实现了，读者可以回顾一下，从第4章开始，到第11章结束，我们首先实现了一条简单的ori指令，通过该指令建立了基本的流水线结构，然后依次添加实现了逻辑操作指令、移位操作指令、空指令、移动操作指令、算术操作指令、转移指令、加载存储指令、协处理器访问指令、异常相关指令，如此由小到大、由简单到复杂，最终实现了教学版OpenMIPS处理器。

从下一章开始将进入本书第三部分——进阶篇，在其中实现了实践版OpenMIPS处理器，并与SDRAM控制器、GPIO模块、Flash控制器、UART控制器、Wishbone总线互联矩阵等模块组成一个小型SOPC，然后下载到FPGA芯片以验证实现效果，最后，还会为实践版OpenMIPS处理器移植嵌入式实时操作系统μC/OS-II。

第三篇 进阶篇

第12章 实践版OpenMIPS处理器设计与实现

第13章 基于实践版OpenMIPS 的小型SOPC的设计与实现

第14章 验证实践版OpenMIPS处理器

第15章 为OpenMIPS处理器 移植μC/OS-II

附录A 教学版OpenMIPS各个 模块的接口说明

附录B OpenMIPS实现的所有 指令及对应的机器码

参考文献

第12章 实践版OpenMIPS处理器设计与实现

经过第4~11章，一步一步地完善、补充，最终实现了我们在第3章中设计的教学版OpenMIPS处理器，但是教学版OpenMIPS处理器距离实用还有一点差距，这也就是实践版OpenMIPS处理器需要解决的问题。

本章将介绍实践版OpenMIPS处理器的设计与实现，首先在12.1节给出了实践版OpenMIPS处理器的设计目标，重点说明实践版OpenMIPS与教学版OpenMIPS的区别，以及添加总线接口的原因。接着在12.2节详细说明Wishbone总线接口，12.3节给出了添加Wishbone总线接口后的实践版OpenMIPS处理器接口图。然后在12.4节介绍实践版OpenMIPS处理器的实现思路。最后，在12.5节通过修改教学版OpenMIPS处理器的代码，实现实践版OpenMIPS处理器。

12.1 实践版OpenMIPS处理器的设计目标

已实现的教学版OpenMIPS处理器的主要设想是尽量简单，比如：在一个时钟周期内可以取到指令，完成存储、加载数据，这样处理器的运行情况（主要是流水线的运行情况）就比较理想化，与教科书相似，代码也很简单清晰，便于使用其进行教学、学术研究和讨论，也有助于学生理解课堂上讲授的知识。所以，我们在构建基于教学版OpenMIPS的最小SOPC时，指令存储器ROM、数据存储器RAM都可

以位于FPGA内部，以满足教学版OpenMIPS的特殊要求。因此，教学版OpenMIPS的接口也比较简单，主要就是与ROM、RAM的接口。如图12-1所示。

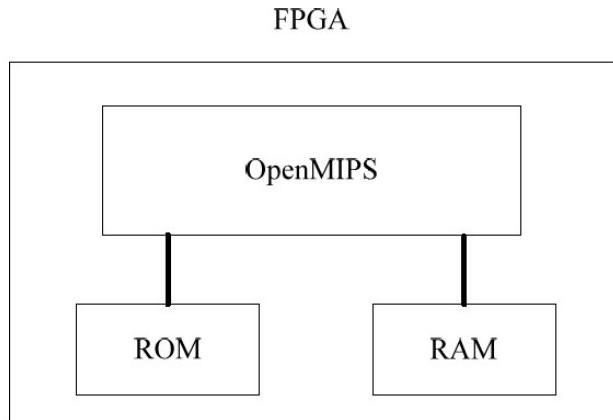


图12-1 基于教学版OpenMIPS的最小SOPC的结构，其指令存储器ROM、数据存储器RAM都位于FPGA内

但是在实际应用中，程序的体积可能非常大，指令存储器就不能再集成在FPGA内部了，一般使用FPGA芯片外部的Flash作为指令存储器。同理，一般使用FPGA芯片外部的SDRAM作为数据存储器，如图12-2所示。

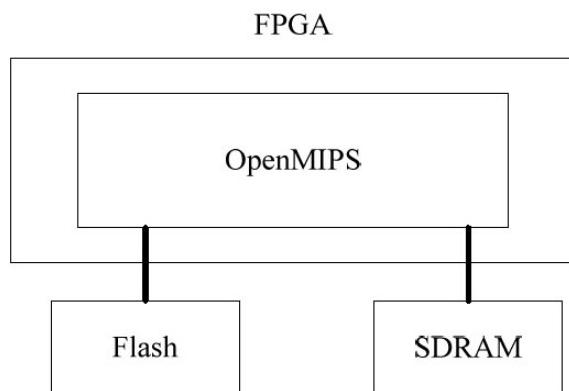


图12-2 实际应用中，指令存储器、数据存储器位于FPGA外部

因此，为了使OpenMIPS实用化，需要为其添加Flash控制器、SDRAM控制器，如图12-3所示。

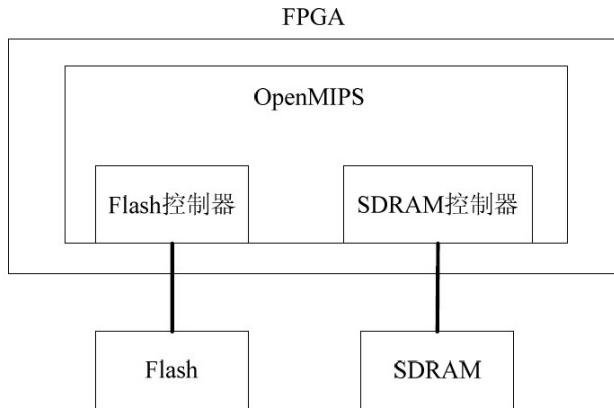


图12-3 为OpenMIPS添加Flash控制器、SDRAM控制器以使其实用化

更进一步，如果要使用FPGA芯片外部的SRAM，那么需要再为OpenMIPS添加SRAM控制器，如果要使用串口，那么需要再为OpenMIPS添加UART控制器。读者一定发现了一个问题：每添加一个外部设备，都要修改OpenMIPS。是的，图12-3的做法不具有良好的扩展性。

参考一下常用的PC，其一般都提供PCI总线接口，各种板卡（包括显卡、语音卡、网卡，甚至用户自制的板卡）只要满足PCI总线接口标准，就可以直接插在PC的PCI插槽使用，十分方便，并不需要修改处理器。借鉴这种方式，我们只需要为OpenMIPS添加通用总线接口，就可以方便地接入新设备。OpenMIPS通过总线接口挂在总线上，各种外部设备的控制器也挂在总线上，如图12-4所示。因为OpenMIPS采用的是哈佛结构，所以有两个总线接口，分别是指令总线接口和数据总线接口。图12-4中的Flash控制器、SDRAM控制器、SRAM控制器、UART控制器都具有同样的总线接口，都可以挂在总线上，并且

都可以放置在FPGA内部。如果要添加其他设备控制器，那么只要具有相同的总线接口，就可以直接挂在总线上，不需要修改OpenMIPS。

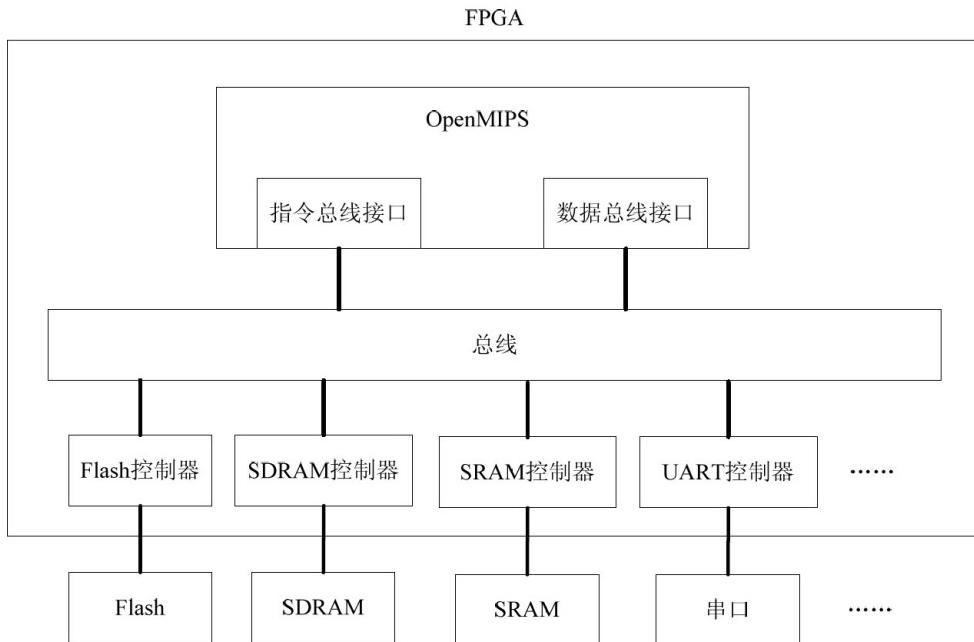


图12-4 改进的连接外部设备的方法——通过总线

各种控制器可以直接使用已有的IP核，目前有很多IP核的研发者或公司，为了方便不同研发者或公司的IP核能够互相连接，就要求这些IP核遵守相同的总线规范。总线规范定义了IP核之间的通用接口。常见的片上总线规范有ARM公司的AMBA、IBM公司的CoreConnect、Altera公司的Avalon，以及Wishbone，本书的实践版OpenMIPS采用的就是Wishbone总线规范。Wishbone总线规范是Silicore公司最先提出的，由于其开放性，现在已有不少用户群，特别是一些开源的IP核，大多数都采用Wishbone规范。

综合上述分析，实践版OpenMIPS处理器的设计目标就是在教学版OpenMIPS处理器的基础上添加Wishbone总线接口，这样就能方便地将其挂接在Wishbone总线上，从而可以使用大量开源的SDRAM、

Flash、GPIO、UART、LCD等模块的控制器，组成一个SOPC，完成特定功能，成为一个能发挥实际作用的处理器。

12.2 Wishbone总线介绍

12.2.1 Wishbone总线接口说明

Wishbone除了开放、免费，还有简单、灵活、轻量、支持用户自定义标签等特点。目前已发布B4版本的规范，OpenMIPS处理器遵循的是Wishbone B2版本的规范。

Wishbone有多种连接方式：点对点、数据流、共享总线、交叉互连等。在点对点连接方式中，有一个主设备，一个从设备，连接关系如图12-5所示，注意图中输出信号的名称使用“_O”结束，输入信号的名称使用“_I”结束。此外，所有的信号都是高电平有效。

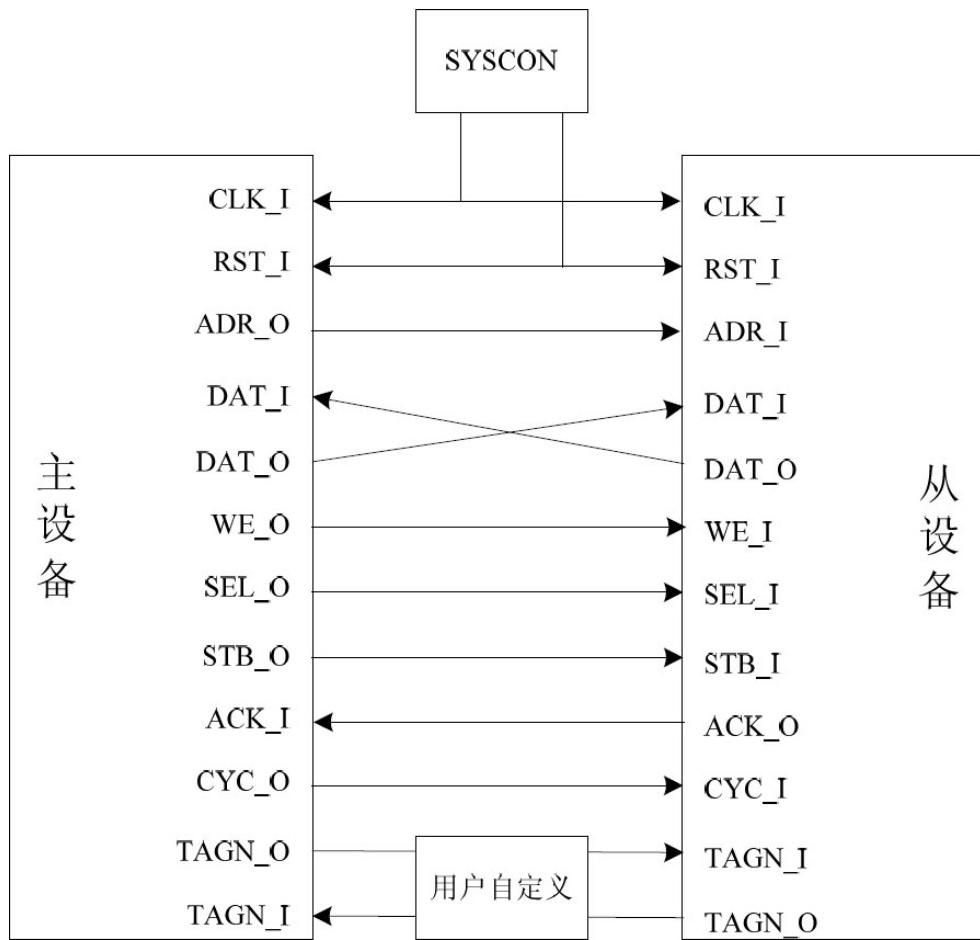


图12-5 Wishbone总线规范点对点连接方式

图12-5中，主、从设备的接口含义如下。

(1) CLK_I、RST_I：分别是时钟信号、复位信号，由外部输入。

(2) DAT_O/DAT_I：数据总线，数据可以由主设备传送给从设备，也可以由从设备传送给主设备。一对主设备和从设备之间最多存在两条数据总线，一条用于主设备向从设备传输数据，另一条用于从设备向主设备传输数据。

(3) ADR_O/ADR_I：地址总线，地址由主设备传送给从设备。

(4) WE_O/WE_I：写使能信号，由主设备传送给从设备，代表当前进行的是写操作还是读操作，1代表写操作，0代表读操作。

(5) SEL_O/SEL_I：数据总线选择信号，用于标识当前操作中，数据总线上哪些比特是有效的，以总线粒度为单位。SEL_O/SEL_I的宽度为数据总线宽度除以数据总线粒度。比如一个具有32位宽、粒度为1个字节的数据总线的选择信号应定义为SEL_O(3:0)/ SEL_I(3:0)，此时，SEL_O(4'b1001)就代表当前操作中数据总线的最高和最低字节有效。

(6) CYC_O/CYC_I：总线周期信号，CYC_O/CYC_I有效代表一个主设备请求总线使用权或者正在占有总线，但是不一定正在进行总线操作（是否正在进行总线操作取决于选通信号STB_O/STB_I是否有效）。只有在CYC_O/CYC_I信号有效的情况下，Wishbone主设备和从设备之间的其他信号才有意义。CYC_O/CYC_I信号在一次总线操作过程中必须持续有效，比如：一次块读操作可能需要多个时钟周期，那么CYC_O/CYC_I信号必须在这多个时钟周期持续有效。

(7) STB_O/STB_I：选通信号。选通信号有效代表主设备发起一次总线操作。只有选通信号有效时（此时CYC_O/CYC_I也必须有效），ADR_O/ADR_I、DAT_O/DAT_I、SEL_O/SEL_I才有意义。

(8) ACK_O/ACK_I：实际还可以有ERR_O/ERR_I、RTY_O/RTY_I，都是主从设备间的操作结束信号。ACK表示成功，ERR表示错误，RTY表示重试。操作总是在某一总线周期内完成的，因此操作结束也称为总线周期结束。成功是操作的正常结束方式，错误表示操作失败，造成失败的原因可能是地址或者数据校验错误，写操作或者读操作不支持等。重试表示从设备当前忙，不能及时处理该

操作，可以稍后重新发起。接收到操作失败或者重试后，主设备如何响应取决于主设备的设计者。

(9) TAGN_O/TAGN_I：标签信号，用户可以利用标签信号传递自定义的信息。

一个总线周期由多个时钟周期构成，用来完成一次操作，可以是单次读/写操作、块读/写操作、读改写操作，总线周期也相应分为单次读/写周期、块读/写周期、读改写周期。本节重点介绍单次读/写周期。

一般情况下，一次操作由主设备和从设备之间的一次握手，以及同时进行的地址和数据总线的一次传输构成。当主设备将CYC_O置高，一个总线周期开始，此后当STB_O为高时，一次总线操作开始。CYC_O和STB_O可以同时从低电平变为高电平，表示开始总线周期的同时发起一次总线操作。

以下分别介绍Wishbone总线单次读操作、单次写操作的过程。

12.2.2 Wishbone总线单次读操作的过程

主从设备之间的信号虽然很多，但单次读/写操作实际上十分简单。单次读操作的Wishbone总线信号如图12-6所示，此处是从主设备的角度观察信号变化情况。

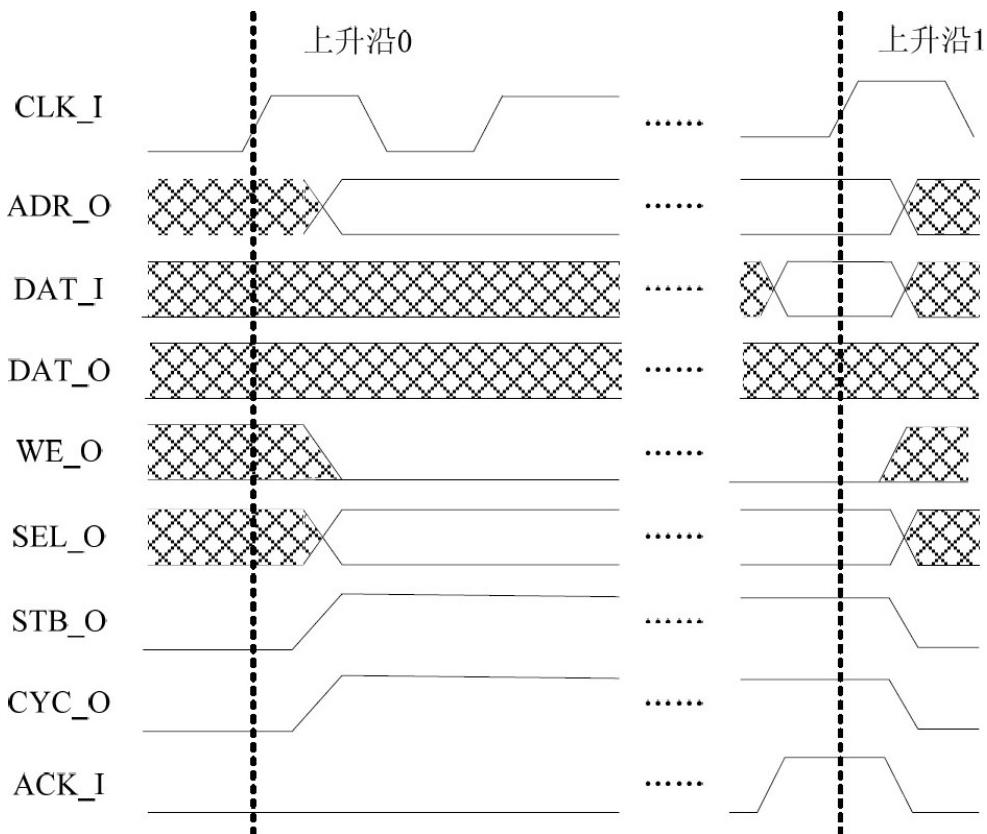


图12-6 Wishbone总线单次读操作时主设备的信号（不考虑TGN_O/TGN_I）

在时钟上升沿0，主设备将地址信号ADR_O、适当的SEL_O放到总线上。将WE_O置低，表示读操作。将CYC_O、STB_O置高表示一次总线操作开始。

在时钟上升沿1到达之前，从设备检测到主设备发起的操作，将适当的数据放到主设备的输入接口DAT_I，同时将主设备的输入ACK_I置高，作为对主设备STB_O的响应。从设备可以在设置ACK_I有效之前，插入任意数量的等待状态。

在时钟上升沿1，主设备发现ACK_I信号为高，于是采样DAT_I信号，作为读取到的数据，并将CYC_O和STB_O置低，表示操作完成。从设备检测到STB_O置低后，将主设备的输入ACK_I也置低。单次读操作就完成了。

12.2.3 Wishbone总线单次写操作的过程

单次写操作的Wishbone总线信号如图12-7所示，此处还是从主设备的角度观察信号变化情况。

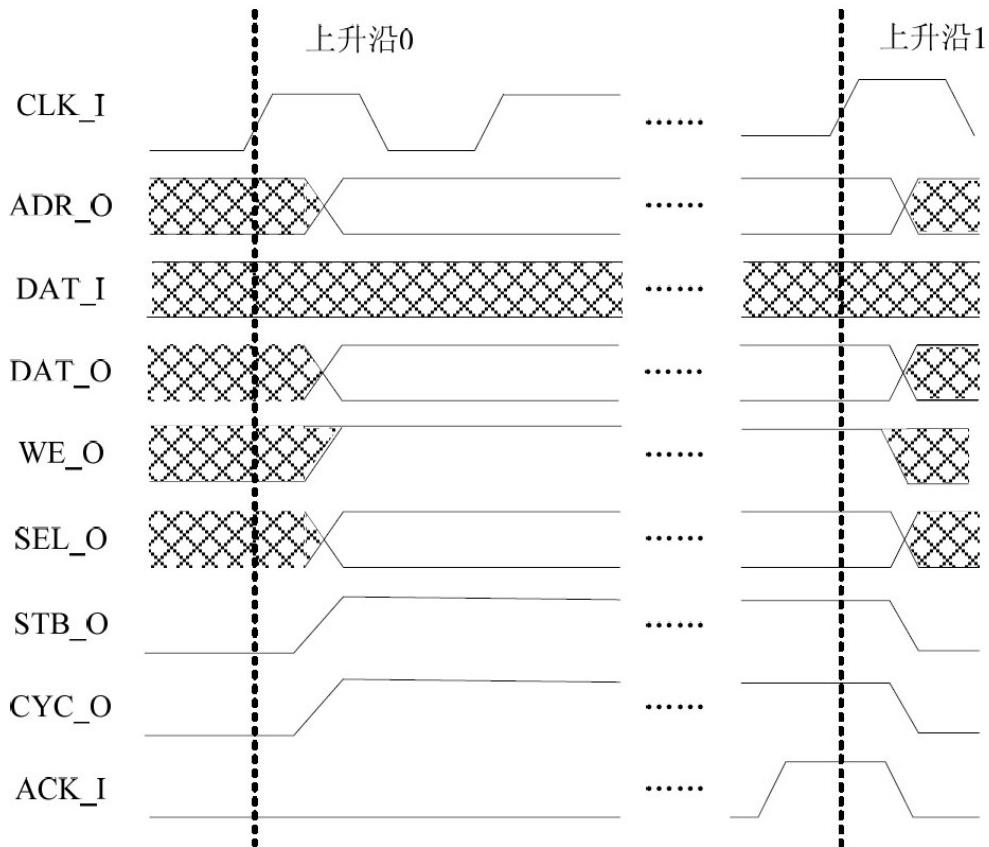


图12-7 Wishbone总线单次写操作时主设备的信号（不考虑TGN_O/TGN_I）

在时钟上升沿0，主设备将地址信号ADR_O、数据信号DAT_O放到总线上，将WE_O置高，表示写操作。将适当的SEL_O放到总线上，以告诉从设备DAT_O中哪些字节是有效的。将CYC_O、STB_O置高，表示一次总线操作开始。

在时钟上升沿1到达之前，从设备检测到主设备发起的操作，于是锁存DAT_O的数据，同时将主设备的输入ACK_I置高，作为对主设备STB_O的响应。从设备可以在设置ACK_I有效之前，插入任意数量的等待状态。

在时钟上升沿1，主设备发现ACK_I信号为高，于是将STB_O和CYC_O置低，表示操作完成。从设备检测到STB_O置低后，将主设备的输入信号ACK_I也置低。单次写操作就完成了。

以上两小节介绍了Wishbone总线规范的单次读/写操作的操作周期，本书不打算介绍Wishbone总线规范的块读/写、读改写操作对应的操作周期，因为实践版OpenMIPS处理器只使用到了单次读/写操作。

12.2.4 SEL_O/SEL_I信号说明

了解了Wishbone总线规范，现在可以解答我们在第9章“加载存储指令的实现”中遇到的一个问题，在9.3.3节解释lb指令的访存过程时，提出如下问题。

“为何不直接设置mem_sel_o为4'b0001，表示希望数据存储器给出的数据的第0-7bit就是要读取的字节，而不考虑mem_addr_i的最后两位为何值，这样不是更简单吗？”

当时的解释如下。

“的确，这样做是更简单了，但是这里确定mem_sel_o值的过程实际上参考了Wishbone总线的相关规范，为了是在后期给OpenMIPS添加Wishbone总线接口的时候容易一些。”

在Wishbone总线规范中，对主设备的输出SEL_O（也就是从设备的输入SEL_I）是有具体规定的，不同的总线宽度、不同的大小端模式，这个规定是不同的。因为OpenMIPS的地址、数据总线宽度是32bit，并且是大端模式，所以此处只介绍当地址、数据总线宽度是32bit，且为大端模式时的规定。

读操作时，主设备通过ADR_O送出32位的地址，从设备将ADR_O最低两位置为0，作为目标地址，给出对应的32位数据，输入到主设备的接口DAT_I。Wishbone总线规范明确了主设备的DAT_I与SEL_O的对应关系，如图12-8所示。



图12-8 读操作时，DAT_I与SEL_O的对应关系

所以，假设lb指令的加载目标地址的最低两位是10，那么就要求设置mem_sel_o为4'b0010。

写操作时，主设备通过ADR_O送出32位的地址，从设备将ADR_O最低两位置为0，作为目标地址。Wishbone总线规范明确了主设备的DAT_O与SEL_O的对应关系如图12-9所示。

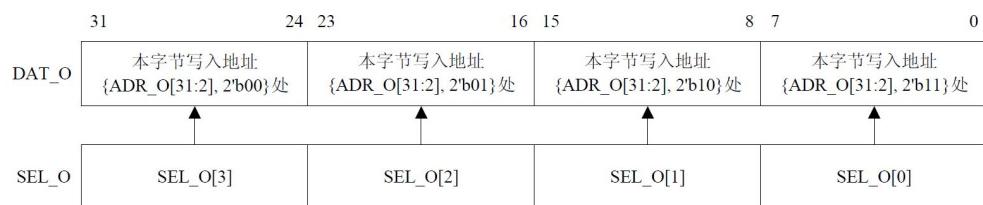


图12-9 写操作时，DAT_O与SEL_O的对应关系

所以，假设sb指令的存储目标地址的最低两位是10，那么就要求设置mem_sel_o为4'b0010。

现在，读者可以回到9.3.3节，体会加载存储指令的访存过程，尤其是其中对信号mem_sel_o的赋值，应该可以理解了。

12.3 实践版OpenMIPS处理器接 口

实践版OpenMIPS处理器仍然采用哈佛结构，即分开的指令、数据接口。其接口如图12-10所示。还是采用左边是输入接口，右边是输出接口的方式绘制，这样比较直观，便于理解。

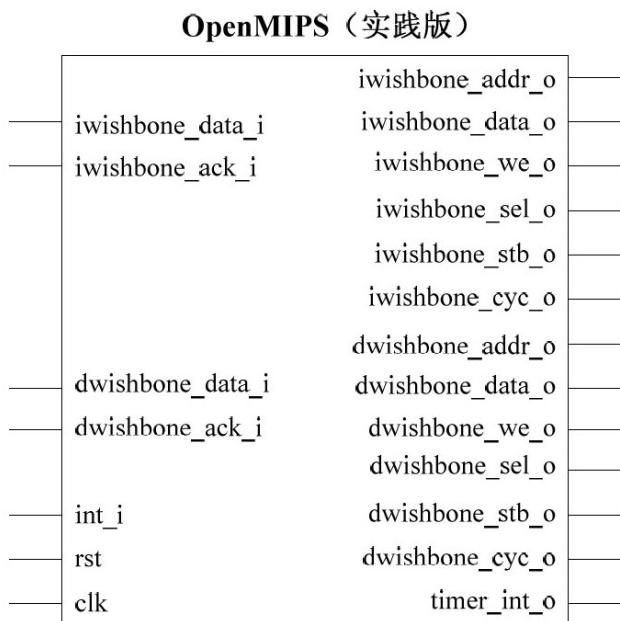


图12-10 实践版OpenMIPS处理器的外部接口图

读者可以对比图3-4教学版OpenMIPS处理器的外部接口图，从中可以发现图12-10只是将图3-4中的指令存储器、数据存储器的接口分

别换成指令Wishbone总线接口、数据Wishbone总线接口，其余未变。各个接口的描述如表12-1所示。

表12-1 实践版OpenMIPS处理器外部接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	int_i	6	输入	6个外部硬件中断输入
4	timer_int_o	1	输出	是否有定时中断发生

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
5	iwishbone_addr_o	32	输出	指令 Wishbone 总线输出的地址
6	iwishbone_data_o	32	输出	指令 Wishbone 总线输出的数据
7	iwishbone_we_o	1	输出	指令 Wishbone 总线写使能信号
8	iwishbone_sel_o	4	输出	指令 Wishbone 总线字节选择信号
9	iwishbone_stb_o	1	输出	指令 Wishbone 总线选通信号
10	iwishbone_cyc_o	1	输出	指令 Wishbone 总线周期信号
11	iwishbone_data_i	32	输入	指令 Wishbone 总线输入的数据
12	iwishbone_ack_i	1	输入	指令 Wishbone 总线输入的响应
13	dewishbone_addr_o	32	输出	数据 Wishbone 总线输出的地址
14	dewishbone_data_o	32	输出	数据 Wishbone 总线输出的数据
15	dewishbone_we_o	1	输出	数据 Wishbone 总线写使能信号
16	dewishbone_sel_o	4	输出	数据 Wishbone 总线字节选择信号
17	dewishbone_stb_o	1	输出	数据 Wishbone 总线选通信号
18	dewishbone_cyc_o	1	输出	数据 Wishbone 总线周期信号
19	dewishbone_data_i	32	输入	数据 Wishbone 总线输入的数据
20	dewishbone_ack_i	1	输入	数据 Wishbone 总线输入的响应

12.4 实践版OpenMIPS处理器的实现思路

实现思路很直观：将教学版OpenMIPS对指令存储器、数据存储器的访问信号分别经过Wishbone总线接口模块，转化为标准的Wishbone总线接口信号，即可。如图12-11所示。

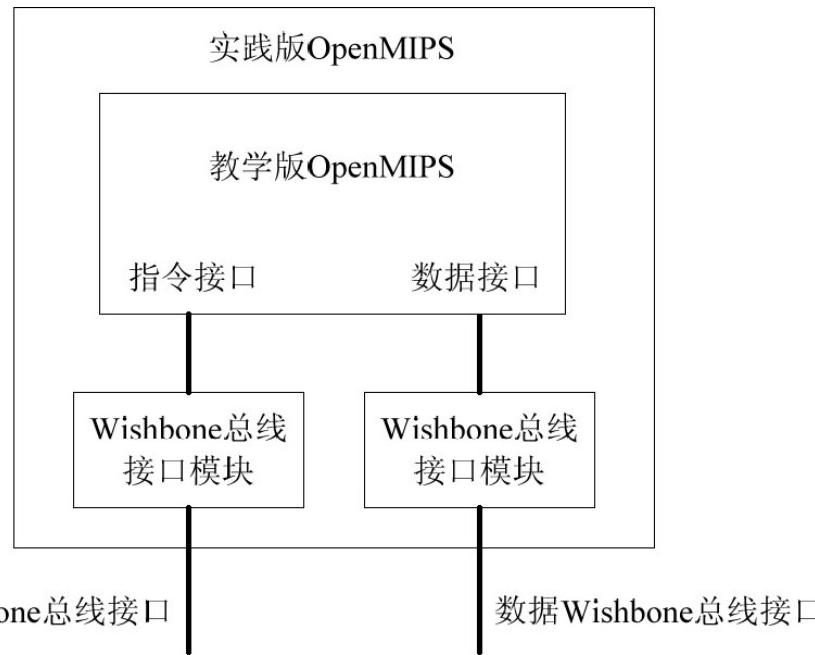


图12-11 实践版OpenMIPS处理器的实现思路

为了实现实实践版OpenMIPS，需要对教学版OpenMIPS的系统结构进行修改，主要修改如图12-12所示。

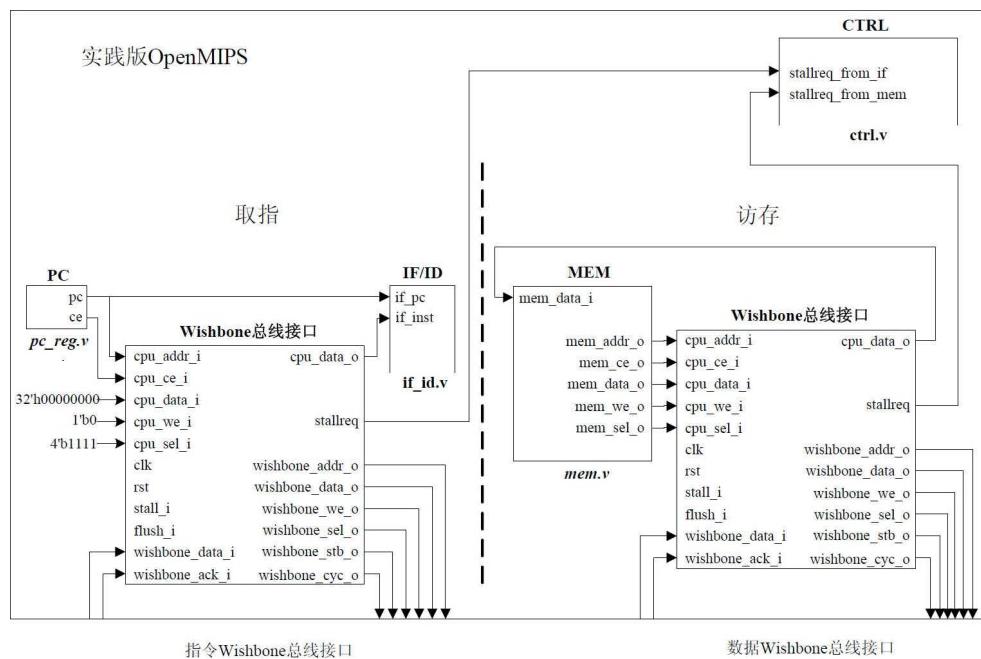


图12-12 在教学版OpenMIPS的基础上进行修改，以实现实实践版OpenMIPS

对图12-12有如下几点说明。

(1) 主要修改了流水线的取指、访存阶段。

(2) 在取指阶段添加了Wishbone总线接口模块，使得PC模块给出的指令存储器访问信号不再直接连接外部指令存储器，而是经过Wishbone总线接口模块转化为Wishbone总线接口信号。

(3) 由于指令存储器是只读的，并且指令宽度固定为32位，所以取指阶段添加的Wishbone总线接口模块的输入cpu_data_i直接设置为32'h00000000；cpu_we_i固定为1'0，表示始终是读操作；cpu_sel_i固定为4'b1111。

(4) 添加Wishbone总线接口后，指令会存储在FPGA芯片外部的Flash中，导致取指时间多于1个时钟周期。在指令没有取到时，需要暂停流水线，所以在取指阶段添加的Wishbone接口模块有一个输出接口stallreq，连接到CTRL模块新增加的输入接口stallreq_from_if，该信号表示取指阶段是否请求流水线暂停。

(5) 在访存阶段也添加了Wishbone总线接口模块，使得MEM模块对数据存储器的访问信号不再直接连接外部数据存储器，而是经过Wishbone总线接口模块转化为Wishbone总线接口信号。

(6) 添加Wishbone总线接口后，数据会存储在FPGA芯片外部的SDRAM中，导致访问数据的时间多于1个时钟周期。在数据没有访问到时，需要暂停流水线，所以在访存阶段添加的Wishbone接口模块也有一个输出接口stallreq，连接到CTRL模块新增加的输入接口stallreq_from_mem，该信号表示访存阶段是否请求流水线暂停。

12.5 从教学版OpenMIPS到实践版OpenMIPS

12.5.1 Wishbone总线接口模块的实现

Wishbone总线接口模块的作用是将处理器对外部设备的访问请求转化为Wishbone总线信号，在图12-12中已经给出了其接口示意图，具体描述如表12-2所示。

表12-2 Wishbone总线接口模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	stall_i	6	输入	CTRL 模块传入的流水线暂停信号
4	flush_i	1	输入	CTRL 模块传入的流水线清除信号
5	cpu_ce_i	1	输入	来自处理器的访问请求信号
6	cpu_data_i	32	输入	来自处理器的数据
7	cpu_addr_i	32	输入	来自处理器的地址信号
8	cpu_we_i	1	输入	来自处理器的写操作指示信号
9	cpu_sel_i	4	输入	来自处理器的字节选择信号
10	cpu_data_o	32	输出	输出到处理器的数据
11	wishbone_addr_o	32	输出	Wishbone 总线输出的地址
12	wishbone_data_o	32	输出	Wishbone 总线输出的数据
13	wishbone_we_o	1	输出	Wishbone 总线写使能信号
14	wishbone_sel_o	4	输出	Wishbone 总线字节选择信号
15	wishbone_stb_o	1	输出	Wishbone 总线选通信号
16	wishbone_cyc_o	1	输出	Wishbone 总线周期信号
17	wishbone_data_i	32	输入	Wishbone 总线输入的数据
18	wishbone_ack_i	1	输入	Wishbone 总线输入的响应
19	stallreq	1	输出	请求流水线暂停的信号

本节将采用有限状态机实现Wishbone总线接口模块，首先定义三个状态：空闲状态（WB_IDLE）、总线忙状态（WB_BUSY）、等待暂停结束状态（WB_WAIT_FOR_STALL）。这三个状态对应的宏定义在defines.v文件中定义，如下。

```
`define WB_IDLE          2'b00    //空闲状态  
`define WB_BUSY          2'b01    //总线忙状态  
`define WB_WAIT_FOR_STALL 2'b11  //等待暂停结束状态
```

三个状态之间的转化关系如图12-13所示。

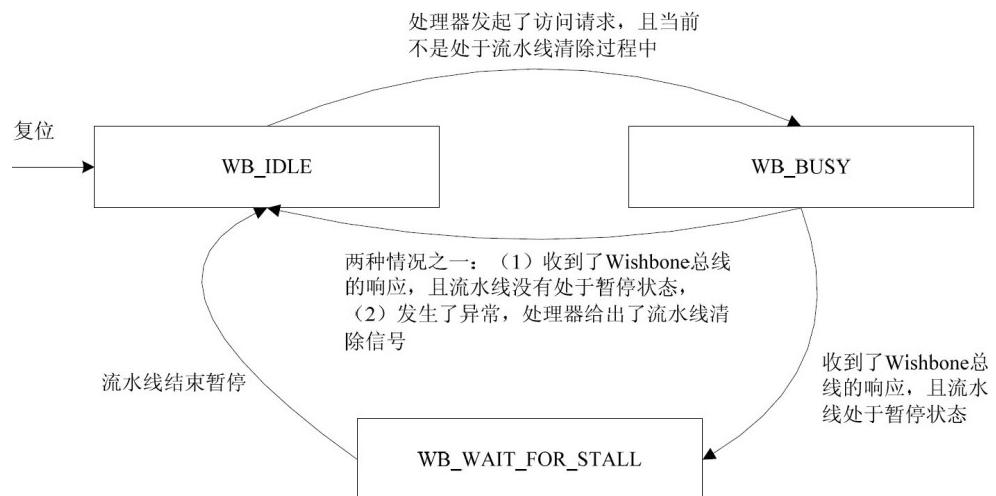


图12-13 Wishbone总线接口模块的状态机

- (1) 复位的时候进入空闲状态WB_IDLE。
- (2) 当处于空闲状态WB_IDLE时，如果处理器发出了访问请求，且当前没有处于流水线清除过程中，那么会进入总线忙状态WB_BUSY，开始访问总线。但是，如果处于流水线清除过程中，那么本次的总线访问当然会无效，所以不必进入WB_BUSY状态。

(3) 当处于总线忙状态WB_BUSY时，如果收到Wishbone总线的响应，表示本次访问结束，此时需要判断流水线是否处于暂停状态。

- 如果没有处于暂停状态，那么将访问到的数据送入处理器，进入空闲状态WB_IDLE，等待下一次访问请求。
- 如果处于暂停状态，那么将访问到的数据暂时保存起来，同时进入等待暂停结束状态WB_WAIT_FOR_STALL。当流水线暂停结束时，再将访问到的数据送入处理器，并且进入空闲状态WB_IDLE，等待下一次访问请求。

(4) 当处于总线忙状态WB_BUSY时，如果发生了异常，那么会清除流水线，此时将直接取消此次Wishbone总线访问，并且回到状态WB_IDLE。

Wishbone总线接口模块的代码如下，源文件是位于本书附带光盘Code\Chapter12目录下的wishbone_bus_if.v文件中。

```
module wishbone_bus_if(  
  
    input wire      clk,  
    input wire      rst,  
  
    //来自ctrl模块  
    input wire[5:0]      stall_i,  
    input wire          flush_i,  
  
    //CPU侧的接口  
    input wire          cpu_ce_i,
```

```

input  wire[`RegBus]          cpu_data_i,
input  wire[`RegBus]          cpu_addr_i,
input  wire                  cpu_we_i,
input  wire[3:0]              cpu_sel_i,
output reg[`RegBus]          cpu_data_o,


//wishbone侧的接口

input  wire[`RegBus]          wishbone_data_i,
input  wire                  wishbone_ack_i,
output reg[`RegBus]          wishbone_addr_o,
output reg[`RegBus]          wishbone_data_o,
output reg                  wishbone_we_o,
output reg[3:0]              wishbone_sel_o,
output reg                  wishbone_stb_o,
output reg                  wishbone_cyc_o,


output reg                  stallreq


);

reg[1:0]      wishbone_state; //保存wishbone总线接口模块的状态
reg[`RegBus]  rd_buf;        //寄存通过Wishbone总线访问到的数据

*****



**



*****
```

```

*****
**/


always @ (posedge clk) begin

    if(rst == `RstEnable) begin

        wishbone_state  <= `WB_IDLE;           //进入WB_IDLE状态
        wishbone_addr_o <= `ZeroWord;
        wishbone_data_o <= `ZeroWord;
        wishbone_we_o   <= `WriteDisable;
        wishbone_sel_o  <= 4'b0000;
        wishbone_stb_o  <= 1'b0;
        wishbone_cyc_o  <= 1'b0;
        rd_buf          <= `ZeroWord;

    end else begin

        case (wishbone_state)

`WB_IDLE:  begin           //WB_IDLE状态

            if((cpu_ce_i == 1'b1) && (flush_i == `False_v))
begin

                wishbone_stb_o  <= 1'b1;
                wishbone_cyc_o  <= 1'b1;
                wishbone_addr_o <= cpu_addr_i;
                wishbone_data_o <= cpu_data_i;


```

```

        wishbone_we_o    <= cpu_we_i;
        wishbone_sel_o   <=  cpu_sel_i;
        wishbone_state   <= `WB_BUSY;      //进入WB_BUSY
状态

        rd_buf           <= `ZeroWord;
end

`WB_BUSY: begin                         //WB_BUSY状态

if(wishbone_ack_i == 1'b1) begin
    wishbone_stb_o  <= 1'b0;
    wishbone_cyc_o  <= 1'b0;
    wishbone_addr_o <= `ZeroWord;
    wishbone_data_o <= `ZeroWord;
    wishbone_we_o    <= `WriteDisable;
    wishbone_sel_o   <= 4'b0000;
    wishbone_state   <= `WB_IDLE;      //进入
WB_IDLE状态

    if(cpu_we_i == `WriteDisable) begin
        rd_buf <= wishbone_data_i;
    end
    if(stall_i != 6'b000000) begin
//进入WB_WAIT_FOR_STALL状态

```

```

        wishbone_state <= `WB_WAIT_FOR_STALL;
    end
end else if(flush_i == `True_v) begin
    wishbone_stb_o  <= 1'b0;
    wishbone_cyc_o <= 1'b0;
    wishbone_addr_o <= `ZeroWord;
    wishbone_data_o <= `ZeroWord;
    wishbone_we_o   <= `WriteDisable;
    wishbone_sel_o  <= 4'b0000;
    wishbone_state   <= `WB_IDLE;           //进入
WB_IDLE状态
    rd_buf          <= `ZeroWord;
end

```

`WB_WAIT_FOR_STALL: begin //WB_WAIT_FOR_STALL状态

```

if(stall_i == 6'b000000) begin
    wishbone_state <= `WB_IDLE;           //进入WB_IDLE
状态
end
end
default: begin
end

```

```

        endcase

    end      //if

end        //always

//*****************************************************************************
** 第二段：给处理器接口信号赋值的组合电路
***** */

***** /

always @ (*) begin
    if(rst == `RstEnable) begin
        stallreq    <= `NoStop;
        cpu_data_o <= `ZeroWord;
    end else begin
        stallreq    <= `NoStop;
        case (wishbone_state)
            `WB_IDLE: begin          //WB_IDLE状态
                if((cpu_ce_i == 1'b1) && (flush_i ==
`False_v)) begin
                    stallreq    <= `Stop;

```

```
        cpu_data_o <= `ZeroWord;
    end
end

`WB_BUSY: begin //WB_BUSY状态

if(wishbone_ack_i == 1'b1) begin
    stallreq <= `NoStop;
    if(wishbone_we_o == `WriteDisable)
begin
    cpu_data_o <= wishbone_data_i;
end else begin
    cpu_data_o <= `ZeroWord;
end
end else begin
    stallreq <= `Stop;
    cpu_data_o <= `ZeroWord;
end
end

`WB_WAIT_FOR_STALL: begin //WB_WAIT_FOR_STALL状态
```

```

        stallreq    <= `NoStop;
        cpu_data_o <= rd_buf;
    end
    default: begin
    end
endcase
end
end
endmodule

```

上述代码可以分为两部分：一部分是控制状态转化的时序电路，另一部分是给处理器接口信号赋值的组合电路。分别解释如下。

(1) 控制状态转化的时序电路

- 复位的时候进入WB_IDLE状态，同时将Wishbone总线的选通信号wishbone_stb_o、周期选择信号wishbone_cyc_o都设置为0，表示无效。
- 在WB_IDLE状态下，如果处理器要访问总线（cpu_ce_i为1），且没有处于流水线清除过程中（flush_i为False_v），那么会进入WB_BUSY状态，同时Wishbone总线的选通信号wishbone_stb_o、周期选择信号wishbone_cyc_o都设置为1，表示开始Wishbone总线访问周期。要访问的地址wishbone_addr_o就是输入信号cpu_addr_i的值，访问类型wishbone_we_o就是输入信号cpu_we_i的值，字节选择信号wishbone_sel_o就是输入信号cpu_sel_i的值。如果是写操作，

那么要写的数据wishbone_data_o就是输入信号cpu_data_i的值。

- 在WB_BUSY状态下，如果收到Wishbone总线的响应(wishbone_ack_i为1)，那么设置Wishbone总线的选通信号wishbone_stb_o、周期选择信号wishbone_cyc_o都为0，从而结束Wishbone总线访问周期。如果是读操作(cpu_we_i为WriteDisable)，那么还会将读到的数据保存到变量rd_buf中。接下来有两种可能：如果流水线没有暂停(stall_i等于6'b0000000)，那么进入空闲状态WB_IDLE。如果流水线暂停(stall_i不等于6'b0000000)，那么进入等待暂停结束状态WB_WAIT_FOR_STALL。
- 在WB_BUSY状态下，如果在还没有收到Wishbone总线的响应时，发生了异常，导致处理器要清除流水线(flush_i为True_v)，那么设置Wishbone总线的选通信号wishbone_stb_o、周期选择信号wishbone_cyc_o都为0，从而结束总线访问周期。同时，回到空闲状态WB_IDLE。
- 在等待暂停结束状态WB_WAIT_FOR_STALL下，当流水线暂停结束时(stall_i等于6'b0000000)，进入空闲状态WB_IDLE。

(2) 给处理器接口信号赋值的组合电路

- 在WB_IDLE状态下，如果处理器要访问总线(cpu_ce_i为1)，且没有处于流水线清除过程中(flush_i为False_v)，那么需要暂停流水线以等待此次Wishbone总线访问结束，所以设置输出信号stallreq为Stop。

- 在WB_BUSY状态下，如果收到Wishbone总线的响应(wishbone_ack_i为1)，那么表示此次访问结束，流水线可以继续，所以设置输出信号stallreq为NoStop。如果是读操作(cpu_we_i为WriteDisable)，那么将读到的数据wishbone_data_i通过cpu_data_o接口送给处理器。
- 在WB_BUSY状态下，如果没有收到Wishbone总线的响应(wishbone_ack_i不为1)，那么表示此次访问还没有结束，流水线要保持暂停，所以设置输出信号stallreq为Stop。
- 在等待暂停结束状态WB_WAIT_FOR_STALL下，此时的Wishbone总线访问已经结束，所以设置输出信号stallreq为NoStop，表示没有由于Wishbone总线访问导致处理器暂停。同时，将缓存到变量rd_buf的数据通过cpu_data_o接口送给处理器。

12.5.2 修改CTRL模块

从图12-12可知，实际例化了两个Wishbone总线接口模块，分别位于流水线的取指阶段、访存阶段。这是由于OpenMIPS处理器设计采用哈佛结构，即分开的指令、数据总线。两个模块各有一个流水线暂停请求信号stallreq，都输出到CTRL模块，分别表示取指阶段请求流水线暂停、访存阶段请求流水线暂停，所以要修改CTRL模块，添加部分接口，如表12-3所示。

表12-3 CTRL模块增加的接口

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	stallreq_from_if	1	输入	取指阶段是否请求流水线暂停
2	stallreq_from_mem	1	输入	访存阶段是否请求流水线暂停

修改CTRL模块的代码如下，修改的部分使用加粗、斜体标识。
完整代码请参考本书附带光盘中Code\Chapter12目录下的ctrl.v文件。

```
module ctrl(  
    .....  
  
    // 新增的输入信号  
    input wire stallreq_from_if,  
    input wire stallreq_from_mem,  
    .....  
);  
  
always @ (*) begin  
    if(rst == `RstEnable) begin  
        .....  
    end else if(excepttype_i != `ZeroWord) begin  
        .....  
    end else if(stallreq_from_mem == `Stop) begin // 访存阶段请求暂停
```

```
    stall <= 6'b011111;
    flush <= 1'b0;

end else if(stallreq_from_ex == `Stop) begin
    stall <= 6'b001111;
    flush <= 1'b0;
end else if(stallreq_from_id == `Stop) begin
    stall <= 6'b000111;
    flush <= 1'b0;

end else if(stallreq_from_if == `Stop) begin // 取指阶段请求暂停
    stall <= 6'b000111;
    flush <= 1'b0;

end else begin
    stall <= 6'b000000;
    flush <= 1'b0;
    new_pc <= `ZeroWord;
end    //if
end      //always
```

```
endmodule
```

读者回顾一下第7章实现暂停机制时的说明，主要有两点。

(1) OpenMIPS采用的是一种改进的流水线暂停机制：假如流水线第n阶段请求流水线暂停，那么需使取指令地址PC的值不变，同时保持流水线第n阶段、第n阶段之前各个阶段的寄存器的值不变，而第n阶段后面的指令继续运行。比如：流水线执行阶段请求流水线暂停，那么保持PC不变，保持取指、译码、执行阶段的寄存器不变，但是允许访存、回写阶段的指令继续运行。

(2) CTRL模块的输出信号stall是一个宽度为6的信号，其含义如下，分别输出到流水线各个阶段。

- stall[0]表示取指地址PC保持不变。
- stall[1]表示流水线取指阶段暂停。
- stall[2]表示流水线译码阶段暂停。
- stall[3]表示流水线执行阶段暂停。
- stall[4]表示流水线访存阶段暂停。
- stall[5]表示流水线回写阶段暂停。

理解了上面的两点就比较容易理解我们对CTRL模块的修改了，还是分取指、访存两种情况解释。

(1) 如果是访存阶段请求暂停，那么除回写阶段外，其余各阶段都要暂停，所以设置stall为6'b011111。

(2) 如果是取指阶段请求暂停, 那么理论上应该只暂停取指阶段、保持PC不变, 也就是设置stall为6'b0000011, 但是上面CTRL模块的代码将stall设置为6'b000111, 使得流水线译码阶段也暂停, 这么做, 主要是考虑到一种特殊情况: 假设译码阶段的指令是转移指令, 那么此时取指阶段将要取到的指令就是延迟槽指令, 将译码阶段也暂停, 保持了转移指令与延迟槽指令在流水线中的相对位置, 从而能够正确识别出延迟槽指令, 如图12-14所示。如果取指阶段暂停, 而不使译码阶段暂停, 那么转移指令会在下一周期进入执行阶段, 同时在译码阶段会填充空指令, 这样就使得填充的空指令被误认为是延迟槽指令, 从而出错, 如图12-15所示。

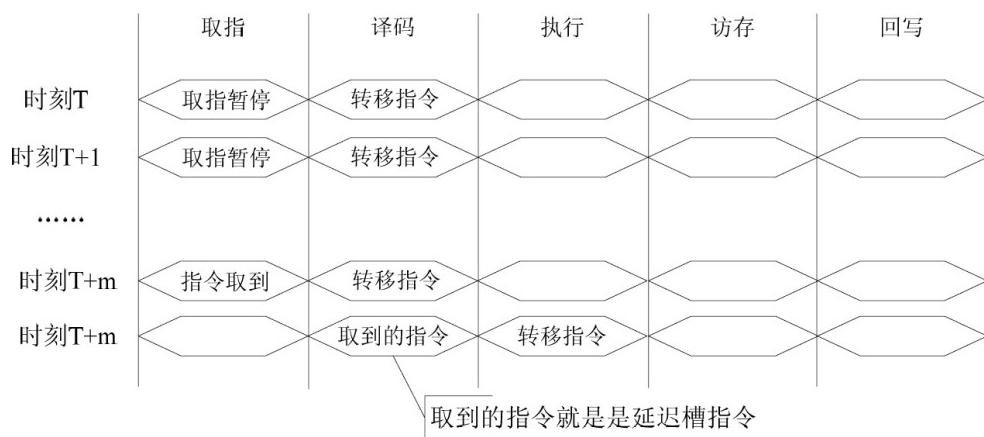


图12-14 取指、译码阶段同时暂停, 从而正确识别延迟槽指令

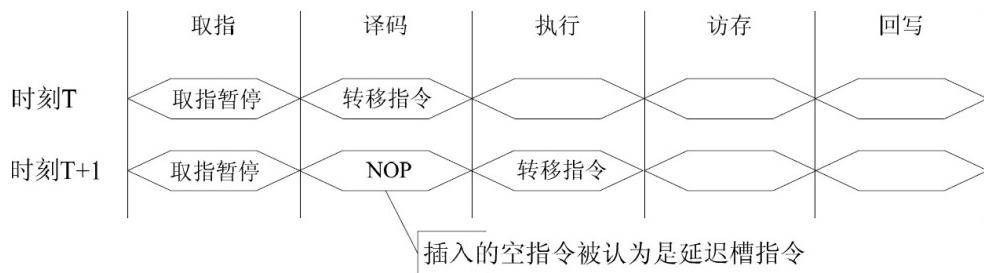


图12-15 取指阶段暂停, 译码阶段没有暂停, 会错误识别延迟槽指令

12.5.3 修改OpenMIPS顶层模块

因为添加了Wishbone总线接口模块，所以需要修改OpenMIPS顶层模块，在其中例化Wishbone总线接口模块，并按照图12-12所示将其和其他模块连接起来，具体代码不在书中罗列，读者可以参考本书附带光盘Code\Chapter12目录下的openmips.v文件。

12.6 实践版OpenMIPS处理器实现小结

本章在教学版OpenMIPS处理器的基础上，通过添加Wishbone总线接口模块，实现了实践版OpenMIPS处理器，这样我们的OpenMIPS处理器就可以方便地使用现在已有的大量开源IP核，包括SDRAM控制器、Flash控制器、UART控制器等，从而可以快速搭建一个实用的SOPC。

在第13章，就将基于实践版OpenMIPS处理器构建一个小型SOPC，该SOPC可以下载实际的FPGA开发板上，用来检验实践版OpenMIPS处理器是否实现正确。

第13章 基于实践版OpenMIPS的小型SOPC

第12章通过为教学版OpenMIPS处理器添加Wishbone总线接口实现了实践版OpenMIPS处理器，但是没有经过验证，本章将基于实践版OpenMIPS搭建一个小型SOPC，下一章将以其为平台，运行验证程序，以检验实践版OpenMIPS处理器是否实现正确。

本章搭建的小型SOPC包括GPIO模块、UART控制器、Flash控制器、SDRAM控制器等，这些控制器与OpenMIPS处理器都连接到Wishbone总线互联矩阵上，本章13.3至13.6节将分别介绍这些控制器。

13.1 小型SOPC的结构

实践版OpenMIPS处理器与其余各种控制器都是通过Wishbone总线连接在一起的，Wishbone总线有四种互联方式：点对点、数据流、共享总线、交叉互联。在点对点方式中，一般只有一个主设备、一个从设备，但是对于一个片上系统而言，一般存在多个模块，并且某一模块能够访问其余多个模块，比如存在CPU、DMA控制器、Flash控制器、SDRAM控制器、GPIO等，其中CPU、DMA控制器作为主设备，Flash控制器、SDRAM控制器、GPIO作为从设备，主设备CPU可以访问所有的从设备，主设备DMA控制器也可以访问所有的从设备，当两者对同一设备发出访问请求时，就需要一个仲裁机制来判断哪个主设备占用总线，所以，片上系统一般使用共享总线或者交叉互联方式。分别介绍如下。

1. 共享总线

共享总线方式适合于系统中有两个或者多个主设备需要与一个或者多个从设备通信的情况，它们通过共享的总线进行通信。主设备在需要与一个从设备通信时，需要先向仲裁器申请总线占有权，获得允许后开始占用总线并与目标从设备开始通信，通信结束后释放总线。当多个主设备同时希望占有总线时，仲裁器通过一定的优先级逻辑分配总线使用机会。其典型框图如图13-1所示。共享总线的缺点是同一时刻只能有一对主、从设备建立通信。

2. 交叉互联

交叉互联主要使用在多个主设备同时访问多个从设备的情况，其典型框图如图13-2所示。在这种连接方式下，主设备发出对某个从设备的访问请求，仲裁器查看总线和从设备是否空闲，从而决定是否给主设备总线访问权。交叉互联方式允许多对主设备和从设备同时进行通信，而共享总线方式在同一时刻只允许一对主、从设备进行通信。

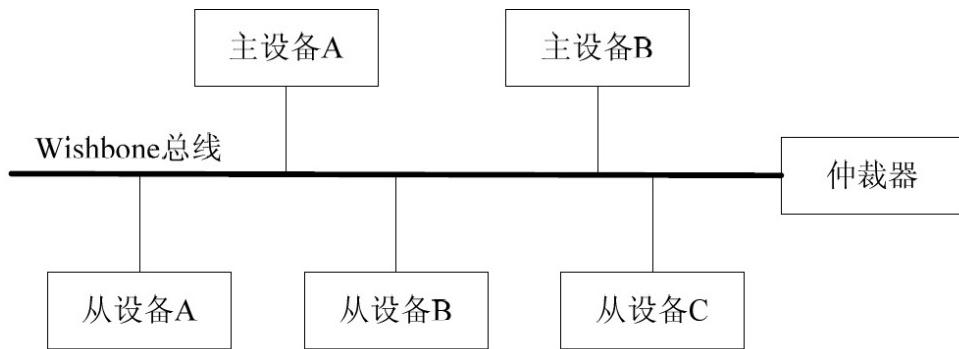


图 13-1 共享总线方式

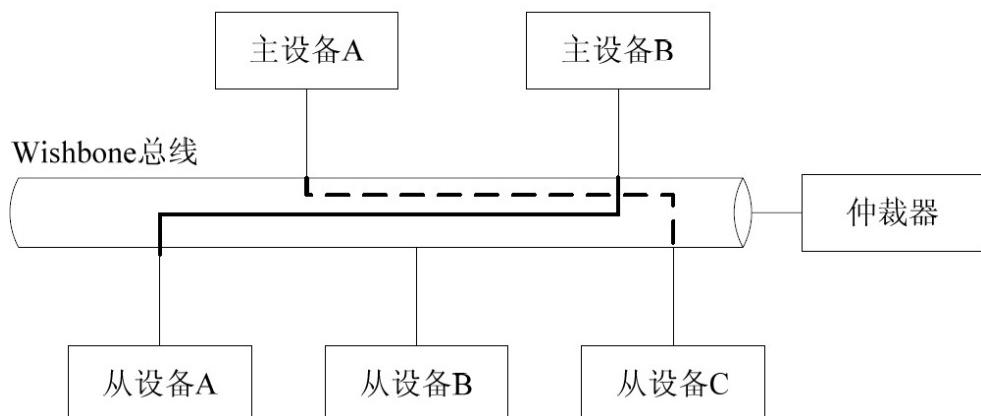


图13-2 交叉互联方式

本章建立的小型SOPC使用的就是交叉互联方式，其结构如图13-3所示。在Wishbone总线上挂接了五个模块：实践版OpenMIPS处理器、GPIO、UART控制器、Flash控制器、SDRAM控制器。其中Wishbone总线使用的是OpenCores站点提供的开源项目WB_CONMAX，这是一个Wishbone总线互联矩阵，采用的是交叉互联方式，允许多对主从设备同时进行通信。

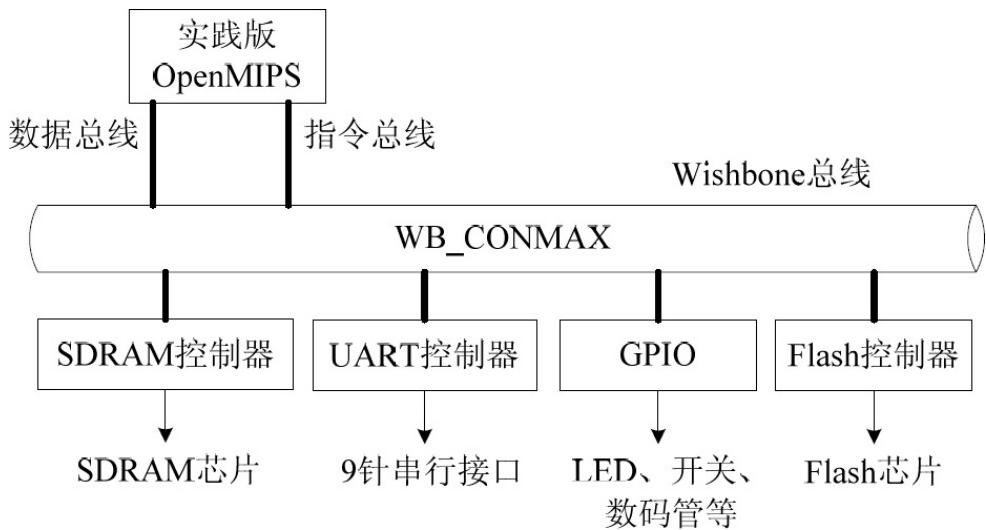


图13-3 小型SOPC的结构

13.2 Wishbone总线互联矩阵 WB_CONMAX

读者可以使用 SVN 从 http://opencores.org/ocsvn/wb_conmax/wb_conmax 下载得到最新的 WB_CONMAX 的代码，下载前需要首先在 OpenCores 站点注册，下载的时候输入在 OpenCores 站点注册时的用户名和密码。也可以在本书光盘 Code\Chapter13\wb_conmax 目录下找到所有代码。WB_CONMAX 模块有如下特点。

- 支持 8 个 Wishbone 总线主设备。
- 支持 16 个 Wishbone 总线从设备。
- 内置仲裁器，支持 1、2 或 4 个优先级。
- 允许多对主从设备同时相互通信。
- 支持 Wishbone B2 版本。

WB_CONMAX模块的结构如图13-4所示。主设备选择从设备进行通信时，依据主设备提供的Wishbone地址的高4位确定是选择哪一个从设备进行通信，当地址高4位为0时，选择的就是从设备0，当地址高4位为15时，选择的就是从设备15。所以每个从设备的寻址空间大小都是256M，其中从设备0的寻址空间是0x00000000-0xFFFFFFFF。本章建立的小型SOPC中各个模块与WB_CONMAX的连接如图13-5所示。

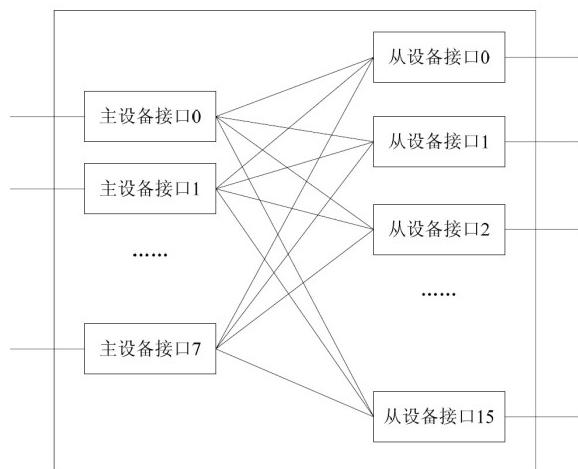


图13-4 WB_CONMAX模块的结构图

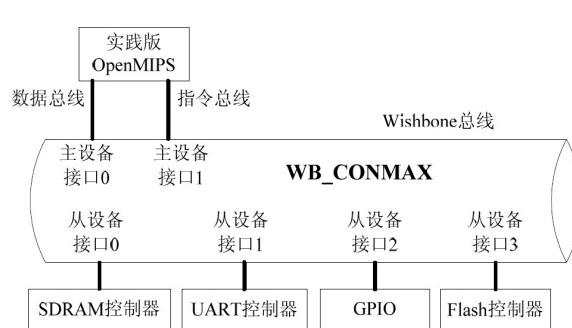


图 13-5 小型 SOPC 中 各 个 模 块 与
WB_CONMAX的连接关系图

OpenMIPS具有分开的指令、数据接口，所以占用WB_CONMAX两个主设备接口，其中数据接口连接到主设备接口0，指令接口连接到主设备接口1。

SDRAM控制器连接到从设备接口0、UART控制器连接到从设备接口1、GPIO连接到从设备接口2、Flash控制器连接到从设备接口3。所以上述各个外设的寻址空间如表13-1所示。

表13-1 小型SOPC中各个外设的寻址空间

名称	寻址空间
----	------

SDRAM	0x00000000-0xFFFFFFFF
UART	0x10000000-0x1FFFFFFF
GPIO	0x20000000-0x2FFFFFFF
Flash	0x30000000-0x3FFFFFFF

一个实用的片上系统通常将程序放置在Flash中，系统启动后从Flash读取第一条指令，从表13-1可知，Flash对应的是从地址0x30000000开始的256M字节空间，所以需要修改实践版OpenMIPS处理器，使得其在复位结束后从地址0x30000000处开始取指。只需要修改取指阶段的PC模块即可实现此目的，主要修改如下，完整代码请读者参考本书附带光盘Code\Chapter13\OpenMIPS目录下的pc_reg.v文件。

```
module pc_reg(
    .....
);

    always @ (posedge clk) begin
        if (ce == `ChipDisable) begin
            pc <= 32'h30000000;           // 取得的第一条指令地址为
            0x30000000
        end
    end
endmodule
```

```
end else begin  
    .....  
endmodule
```

13.3 GPIO

GPIO (General Purpose Input Output) 是以位为单位进行数字输入输出的I/O接口，作为单纯的通用输入/输出I/O，输入时从外部读取输入信号，输出时将写入的值输出到外部。处理器通过GPIO可以与各种设备相连接，例如：LED、开关、七段数码管等。

本章建立的小型SOPC将直接使用OpenCores站点提供的开源项目GPIO IP Core，读者可以在OpenCores站点下载源代码，也可以直接在本书附带光盘Code\Chapter13\gpio目录下找到所有源代码。在本书附带光盘中Doc目录下提供了该GPIO的说明手册。其具有如下特点。

- I/O接口数量从1到32可配置。
- 所有的I/O接口都可以配置为双向接口。
- 输入接口可以触发中断。
- 具有复用输入接口，GPIO最终的输出信号可以是Wishbone总线接口的输入信号，也可以是复用输入接口的信号。
- 可以采用Wishbone总线的时钟，也可以采用单独的时钟。
- 支持Wishbone B版本（手册中未明确说明是哪个版本，从接口情况分析，可能是B2版本）。

接口描述如表13-2所示。

表13-2 GPIO IP核的外部接口描述

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	wb_clk_i	1	输入	Wishbone 总线时钟信号
2	wb_RST_i	1	输入	Wishbone 总线复位信号
3	wb_cyc_i	1	输入	Wishbone 总线周期信号
4	wb_adr_i	7	输入	Wishbone 总线输入的地址
5	wb_dat_i	32	输入	Wishbone 总线输入的数据
6	wb_we_i	1	输入	Wishbone 总线写使能信号
7	wb_sel_i	4	输入	Wishbone 总线字节选择信号
8	wb_stb_i	1	输入	Wishbone 总线选通信号
9	wb_dat_o	32	输出	Wishbone 总线输出的数据
10	wb_ack_o	1	输出	Wishbone 总线输出的响应
11	wb_err_o	1	输出	Wishbone 总线输出的错误响应
12	wb_inta_o	1	输出	中断信号
13	aux_i	1~32	输入	复用输入信号
14	ext_pad_i	1~32	输入	输入信号
15	ext_pad_o	1~32	输出	输出信号
16	ext_padoe_o	1~32	输出	输出使能信号
17	clk_pad_i	1	输入	外部时钟输入

GPIO IP核的功能是通过配置一系列寄存器实现的，主要的寄存器如表13-3所示。

表13-3 GPIO IP核中的主要寄存器

寄存器名称	地 址	宽 度	访问方式	作用描述
RGPIO_IN	Base + 0x0	1~32	只读	输入到 GPIO 的信号
RGPIO_OUT	Base + 0x4	1~32	可读可写	GPIO 输出的信号
RGPIO_OE	Base + 0x8	1~32	可读可写	GPIO 输出接口使能信号
RGPIO_INTE	Base + 0xC	1~32	可读可写	中断使能信号

其中地址一栏中的Base就是GPIO的基地址，从图13-5可知，小型SOPC中的GPIO挂接在从设备接口2，因此GPIO对应的地址空间是0x20000000-0x2FFFFFFF，所以表13-3中的Base就等于0x20000000，这样就可以知道各个寄存器的确切地址了，例如：RGPIO_OUT寄存器的地址就是0x20000004。对其中的RGPIO_OE、RGPIO_INTE两个寄存器解释如下。

(1) RGPI0_OE

宽度为1~32可选，当某一位为1时，相应的输出接口使能，当某一位为0时，相应的输出接口为三态或open-drain模式。复位的时候，所有位都为0。

(2) RGPI0_INTE

宽度为1~32可选，当某一位为1时，相应的输入接口能产生中断，当某一位为0时，相应的输入接口不能产生中断。复位的时候，所有位都为0。

GPIO IP核由 gpioDefines.v、 gpio_top.v 两个文件组成。其中 gpioDefines.v 文件有一些宏定义需要配置，目的是使得输入、输出接口的宽度都是32，如下。

```
//IO接口的数量，可以配置的范围是1-32，默认是31，此处改为32
`define GPIO_IOS 32

//与IO接口数量要对应，默认是31，此处也改为32
`define GPIO_LINES 32

//将下面的宏定义注释掉，表示没有复用输入接口，也就是没有表13-2中的aux_i接口
//`define GPIO_AUX_IMPLEMENT

//在小型SOPC中，GPIO模块的时钟采用的是Wishbone总线的时钟，所以将下面的宏
//定义注释掉，
```

```
//表示没有外部时钟输入接口，也就是没有表13-2中的clk_pad_i接口  
//`define GPIO_CLKPAD
```

13.4 UART控制器

13.4.1 UART简介

UART 即 通 用 异 步 收 发 器 (Universal Asynchronous Receiver/Transmitter) , 是广泛使用的串行数据传输协议。它的功能是将并行的数据转变为串行的数据发送或者将接收到的串行数据转变为并行数据。当本章设计的小型SOPC具备了UART控制器之后，就可以通过串口与计算机进行通信了。

UART产生于20世纪70年代，Intel 8250是第一代产品，被使用在IBM-PC及其兼容机上，用于与Modem或串行打印机进行通信，随着PC的成功和迅速普及，确定了UART的结构和特性，经过8250A、16450、16C451、16550，逐步发展到16550A。16550A与用于8250的软件兼容，但是提供了更高的性能，其性能增强的关键是使用了先进先出堆栈（FIFO）作为缓存，配置了16字节的发送FIFO和16字节的接收FIFO。之后虽然也有新的UART出现，但是从内部结构分析，实际都是在16550的基础上增加了寄存器，从软件角度分析，变化并不大。因此，16550是实际上的工业标准UART，分为A、B、C、D共4种型号。

下面分别介绍UART数据传输、数据接收、流控制的基本过程。

(1) UART的数据传输

UART在传输的时候，将待传输数据的每个字符一位一位地传输。传输格式如图13-6所示。

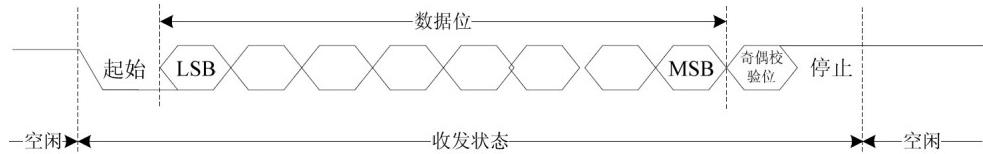


图13-6 UART的数据传输格式

依次传输起始位、数据位、奇偶校验位、停止位，分别说明如下。

起始位：先发出一个低电平信号，也就是逻辑“0”，表示传输的开始。

数据位：紧接着起始位之后的是数据位。数据位的个数可以是4、5、6、7、8等，构成一个字符，从字符的最低位开始传送。

奇偶校验位：数据位之后是奇偶校验位。数据位加上这一位后，使得“1”的个数为偶数（偶校验）或奇数（奇校验），以此来判断数据传送的正确与否。

停止位：是一个字符数据的结束标志，可以是1位、1.5位、2位的高电平信号。

UART的通信速率用波特率（baud rate）来表示。波特率指的是信号被调制以后的变化率，即单位时间内载波变化的次数。用于波特率计算的信号除了数据位，还包括起始位、奇偶校验位、停止位，因

此，波特率与单纯的数据传输速率是不同的。UART常用的波特率有9600 baud、19200 baud、38400 baud等。

(2) UART的数据接收

UART的数据接收部分采用比波特率高的采样频率实现。实际使用中，一般使用比波特率高16倍的接收时钟进行采样。为了便于说明，图13-7以接收时钟是波特率的4倍为例，给出了数据接收过程。

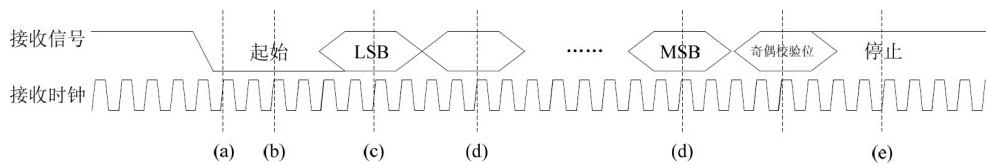


图13-7 UART的数据接收过程

- (a) 当接收信号由高电平变为低电平时，表示检测到起始位。
- (b) 检测到起始位后，在接下来的第2个时钟周期检查接收信号，如果保持为低电平，说明确实是起始位，开始接收数据。否则认为起始位检测错误，将其忽略。
- (c) 确定是起始位后，等待4个时钟周期检查接收信号，得到的值就是接收到的第一个bit，也就是LSB。
- (d) 之后每隔4个时钟周期检查接收信号，依次得到传送过来的数据位、奇偶校验位。从图中可以发现，每次采样都是在接收数据的中部，这样采样得到的数据更加准确。
- (e) 数据接收完成后，接收停止位。

(3) 流控制

数据在两个UART之间传输时，常常会出现丢失数据的现象，比如：两台PC的处理速度不同，如果接收方数据缓冲区已满，那么此时继续发送来的数据就会丢失，特别是使用PC机与Modem进行数据传输时，这个问题尤为突出，流控制就是用于解决这个问题的。当接收方数据处理不过来时，就发出“不再接收”的信号，发送方则停止发送，直到收到“可以继续发送”的信号再发送数据，因此流控制可以控制数据传输进程，防止数据丢失。UART常用的两种流控制是硬件流控制和软件流控制。

对硬件流控制进行简单介绍。引入了两对握手信号RTS/CTS、DTR/DSR。第一对握手信号是RTS和CTS，当接收方准备好接收数据时，将RTS置高，表示它准备好了，如果发送方也就绪，那么置高CTS，表示它即将发送数据。第二对握手信号是DTR和DSR，主要用于Modem通信。例如：当Modem已经准备好接收来自PC的数据时，就置高DTR，表示和电话线的连接已经建立。如果DSR线置高，那么PC将开始发送数据。一个简单的规则是DTR/DSR用于表示系统通信就绪，而RTS/CTS用于单个数据包的传输。

13.4.2 UART16550 IP核介绍

本章建立的小型SOPC采用的UART控制器是OpenCores站点提供的开源项目UART16550 IP Core，读者可以在OpenCores站点下载源代码，也可以直接在本书附带光盘Code\Chapter13\uart目录下找到所有源代码。该IP核具有如下特点。

- 最大程度的兼容国家半导体公司（National Semiconductor）的16550A设备。

- 支持Wishbone B⁴版本。
- Wishbone接口侧的数据总线宽度可以设置为32或者8。

其接口可以分为三部分，分别如表13-4、表13-5、表13-6所示。

表13-4 UART16550 IP核的Wishbone接口信号

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	wb_clk_i	1	输入	Wishbone 总线时钟信号
2	wb_RST_i	1	输入	Wishbone 总线复位信号
3	wb_cyc_i	1	输入	Wishbone 总线周期信号
4	wb_addr_i	3 或者 5	输入	Wishbone 总线输入的地址
5	wb_dat_i	8 或 32	输入	Wishbone 总线输入的数据
6	wb_we_i	1	输入	Wishbone 总线写使能信号

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
7	wb_sel_i	4	输入	Wishbone 总线字节选择信号
8	wb_stb_i	1	输入	Wishbone 总线选通信号
9	wb_dat_o	8 或 32	输出	Wishbone 总线输出的数据
10	wb_ack_o	1	输出	Wishbone 总线输出的响应

表13-5 UART16550 IP核的中断信号

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	int_o	1	输出	中断信号

表13-6 UART16550 IP核的串行接口信号

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	stx_pad_o	1	输出	串口输出
2	srx_pad_i	1	输入	串口输入
3	rts_pad_o	1	输出	RTS (Request To Send) 请求发送信号
4	cts_pad_i	1	输入	CTS (Clear To Send) 清除发送信号
5	dtr_pad_o	1	输出	DTR (Data Terminal Ready) 数据终端准备好信号
6	dsr_pad_i	1	输入	DSR (Data Set Ready) 数据准备好信号
7	ri_pad_i	1	输入	RI (Ring Indicator) 振铃指示信号
8	ded_pad_i	1	输入	DCD (Data Carrier Detect) 数据载波检测信号
9	baud_o	1	输出	波特率输出信号，其频率是实际波特率的 16 倍，该信号是可选的

同GPIO一样，UART16550 IP核的功能也是通过配置一系列寄存器实现的，如表13-7所示。

表13-7 UART16550 IP核中的寄存器

寄存器名称	地址	宽度	访问方式	作用描述
Receiver Buffer	Base + 0x0	8	只读	接收缓冲
Transmitting Holding Register (THR)	Base + 0x0	8	只写	发送保持
Interrupt Enable Register	Base + 0x1	8	可读可写	中断控制
Interrupt Identification	Base + 0x2	8	只读	中断标志
FIFO Control	Base + 0x2	8	只写	FIFO 控制
Line Control Register (LCR)	Base + 0x3	8	可读可写	线路控制
Modem Control	Base + 0x4	8	只写	Modem 控制
Line Status (LS)	Base + 0x5	8	只读	线路状态
Modem Status	Base + 0x6	8	只读	Modem 状态

其中地址一栏中的Base就是UART控制器的基地址，从图13-5可知，小型SOPC中的UART控制器挂接在从设备接口1，因此UART控制器对应的地址空间是0x10000000-0x1FFFFFFF，所以表13-7中的Base就等于0x10000000，于是便可以知道各个寄存器的确切地址。除了表13-7中列出的寄存器之外，还有两个寄存器，组成一个16bit的分频系数，用于时钟分频，如表13-8所示。

表13-8 UART16550 IP核中的分频系数寄存器

寄存器名称	地址	宽度	访问方式	作用描述
Divisor Latch Byte 1	Base + 0x0	8	可读可写	分频系数的 LSB
Divisor Latch Byte 2	Base + 0x1	8	可读可写	分频系数的 MSB

读者朋友可能注意到，此处两个分频系数寄存器的地址与表13-7中的寄存器有冲突，解决方法是，当Line Control Register (LCR) 寄存器的第7bit为1时，地址Base + 0x0、base + 0x1对应的就是两个分频系数寄存器，反之，对应的是表13-7中的寄存器。

本书在第14章验证小型SOPC的时候，没有使用UART16550 IP核的流控制等功能，只是使用了简单的数据发送、接收功能，因此只对与数据发送、接收有关的寄存器进行说明，其余寄存器的说明可以参考本书附带光盘Doc目录下UART16550 IP核的使用手册。需要说明的寄存器是 Interrupt Enable Register（IER）、Line Control Register（LCR）、Line Status（LS）、分频系数寄存器。

（1） IER

中断使能寄存器用来设置是否允许中断，各位的作用如表13-9所示。

表13-9 IER寄存器

位	作用描述
0	数据接收中断：0——禁止；1——使能
1	Transmitting Holding Register空中断：0——禁止；1——使能
2	接收线状态中断：0——禁止；1——使能
3	Modem状态中断：0——禁止；1——使能
4-7	保留

（2） LCR

LCR寄存器用来设置接收、发送的数据格式。各位的作用如表13-10所示。

表13-10 LCR寄存器

位	作用描述
0-1	数据位长度： 00——5位 01——6位 10——7位 11——8位
2	停止位长度： 0——1位停止位 1——2位停止位（当数据位长度为5时，表示1.5位停止位）
3	校验位： 0——无校验 1——有校验
4	校验类型： 0——奇校验 1——偶校验
5	添加奇偶校验位： 0——禁止添加奇偶校验位 1——使能添加奇偶校验位

6	间断控制位： 0——不使用间断 1——串行输出固定在逻辑0
7	分频系数寄存器访问位： 0——访问正常寄存器 1——访问分频系数寄存器

(3) LS

LS寄存器用来指示发送和接收的状态，各个位的作用如表13-11所示。

表13-11 LS寄存器

位	作用描述
0	接收数据标志： 1——接收FIFO不为空 0——接收FIFO为空
1	接收FIFO溢出标志： 1——FIFO满并且接收移位寄存器正在接收新数据。 读LS寄存器后，会自动清除该位 0——接收FIFO没有溢出
2	校验错误标志： 1——FIFO顶部的字节有校验错误。读LS寄存器后， 会自动清除该位 0——当前字节没有校验错误

3	<p>帧错误标志：</p> <p>1——FIFO顶部字节有帧错误。读LS寄存器后，会自动清除该位 0——当前字节没有帧错误</p>
4	<p>中断中断标志：</p> <p>1——当前字节有中断错误。当串行接收信号保持一段时间的逻辑0时，认为发生中断错误，此时写一个0字节到FIFO。读LS寄存器后，会自动清除该位 0——当前字节没有中断错误</p>
5	<p>发送FIFO空标志：</p> <p>1——THR寄存器为空，但是发送移位寄存器不为空。在这种情况下，会产生THR空中断。向发送FIFO写入数据后，会自动清除该位 0——其余情况</p>
6	<p>发送数据空标志：</p> <p>1——THR寄存器和发送移位寄存器都为空。向发送FIFO写入数据后，会自动清除该位 0——其余情况</p>
7	<p>1——FIFO模式时，至少有一个检验错误、帧错误或者中断发生。读LS寄存器后，会自动清除该位 0——其余情况</p>

(4) 分频系数寄存器

两个分频系数寄存器形成一个16bit的分频系数，其值需要依据系统时钟、波特率进行计算，计算方法如下。

$$\text{分频系数} = \text{系统时钟} / (16 \times \text{波特率})$$

使用上式的结果设置分频系数寄存器，而且设置的时候，要先写高字节，也就是将分频系数的高8位写入寄存器Divisor Latch Byte 2，再写低字节，也就是将分频系数的低8位写入寄存器Divisor Latch Byte 1。

读者现在可能对上述寄存器的作用还不太理解，在第14章为小型SOPC编写测试程序的时候，读者会切实体会到这几个寄存器的作用。

13.5 Flash控制器

13.5.1 Flash简介

本章设计的小型SOPC将程序保存在Flash中，并且从Flash启动，所以需要Flash控制器，来进行Flash的读操作。Flash主要有两种：NAND flash、NOR flash，前者以“块”为基本单位进行访问，后者以“字”为基本单位进行访问，因此，程序可以直接在NOR Flash里运行，不必把程序复制到RAM里才运行。也正是这个原因，一般把系统启动代码存放到NOR Flash里，实现从Flash启动系统。本章实现的Flash控制器实际就是NOR Flash的控制器，下文中的Flash也都是指NOR Flash，不再明确指出。

NOR Flash的接口一般如表13-12所示。

表13-12 NOR Flash的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	ADDR	依据 Flash 容量确定	输入	要访问的地址
2	DAT	8、16 或 32	输入/输出	写入的数据/读出的数据
3	CE	1	输入	片选信号, 低电平有效
4	OE	1	输入	输出使能信号, 低电平有效
5	WE	1	输入	写使能信号, 低电平有效
6	RESET	1	输入	复位信号, 低电平有效

Flash的读取速度较低，我们设想的是：小型SOPC将程序存储在Flash中，启动后将主应用程序（比如：操作系统）从Flash复制到SDRAM运行，以加快运行速度，所以只涉及Flash的读操作，本章实现的Flash控制器也只支持读操作。

Flash读操作的时序如图13-8所示。在RESET无效（高电平表示无效）的前提下，ADDR接口给出读地址，CE、OE变为有效（低电平表示有效），WE置为无效（高电平表示无效），开始读操作，等待Tacc时间后，数据通过DAT接口输出。在这个过程中，时间Tacc很关键，其表示从给出读地址到数据输出的时延，不同型号的Flash有不同的Tacc。

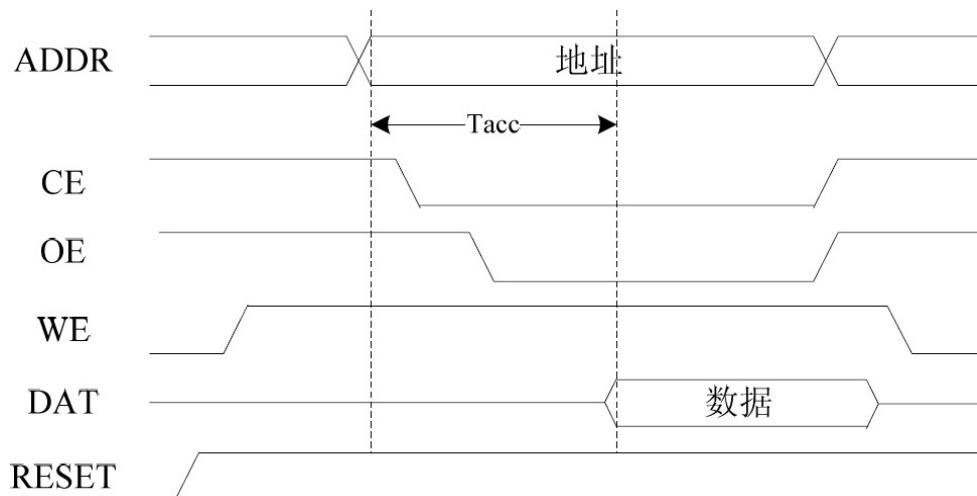


图13-8 Flash读操作的时序

13.5.2 Flash控制器的设计

从图13-8可知，Flash的读操作很简单，设置完成地址信号、片选信号、输出使能信号后，只需等待一段时间Tacc，就可以得到要读取的数据。此处的关键是Tacc的值是多少，该值与具体Flash芯片有关。

本章设计的SOPC将在DE2开发平台上运行，DE2上的Flash芯片是Spansion公司的S29AL032D70TFI04，容量是4MB，是一种NOR Flash。通过查询芯片手册可知Tacc最大值等于70ns。DE2上的时钟有两种：27MHz、50MHz，本章设计的小型SOPC计划采用27MHz的时钟，一个时钟周期大约是37ns，所以最多需要等待3个时钟周期就能得到要读取地址的数据。

此外，S29AL032D70TFI04芯片的数据线宽度是8，当处理器通过Wishbone总线读取指令时，一条指令为32位，所以共需要4次Flash读操作。本书附带光盘Doc目录下有Flash芯片S29AL032D70TFI04的手册。

需要说明一点：虽然本节设计的Flash控制器针对的是S29AL032D70TFI04芯片，但是其他Flash芯片的控制器都可以在此基础上经过修改得到。

13.5.3 Flash控制器的实现

Flash控制器的接口如表13-13所示，可以分为两部分：Wishbone总线接口部分（序号1-10）、Flash芯片接口部分（序号11-16）。

表13-13 Flash控制器的接口

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	wb_clk_i	1	输入	Wishbone 总线时钟信号
2	wb_RST_i	1	输入	Wishbone 总线复位信号
3	wb_cyc_i	1	输入	Wishbone 总线周期信号
4	wb_adr_i	32	输入	Wishbone 总线输入的地址
5	wb_dat_i	32	输入	Wishbone 总线输入的数据
6	wb_we_i	1	输入	Wishbone 总线写使能信号
7	wb_sel_i	4	输入	Wishbone 总线字节选择信号
8	wb_stb_i	1	输入	Wishbone 总线选通信号
9	wb_dat_o	32	输出	Wishbone 总线输出的数据
10	wb_ack_o	1	输出	Wishbone 总线输出的响应
11	flash_adr_o	32	输出	Flash 地址信号
12	flash_dat_i	8	输入	从 Flash 读出的数据
13	flash_RST	1	输出	Flash 复位信号，低电平有效
14	flash_OE	1	输出	Flash 输出使能信号，低电平有效
15	flash_CE	1	输出	Flash 片选信号，低电平有效
16	flash_we	1	输出	Flash 写使能信号，低电平有效

Flash 控制器的代码如下所示，源文件位于本书附带光盘中 Code\Chapter13\flash 目录下的 wb_flash.v 文件中。

```
module flash_top(  
  
    // Wishbone总线接口  
    wb_clk_i, wb_RST_i, wb_adr_i, wb_dat_o,  
    wb_dat_i, wb_sel_i, wb_we_i, wb_stb_i,  
    wb_cyc_i, wb_ack_o,  
  
    // Flash芯片接口  
    flash_adr_o, flash_dat_i, flash_RST,  
    flash_OE,     flash_CE,      flash_we  
);
```

```

input                  wb_clk_i;
input                  wb_rst_i;
input [31:0]           wb_adr_i;
output reg [31:0]      wb_dat_o;
input [31:0]           wb_dat_i;
input [3:0]            wb_sel_i;
input                  wb_we_i;
input                  wb_stb_i;
input                  wb_cyc_i;
output reg             wb_ack_o;
output reg [31:0]      flash_adr_o;
input [7:0]            flash_dat_i;
output                flash_RST;
output                flash_OE;
output                flash_CE;
output                flash_WE;
reg      [3:0]          waitstate;
wire     [1:0]          adr_low;

// 如果Wishbone总线开始操作周期，那么设置变量wb_acc为1;
// 而且，如果是读操作，那么设置变量wb_rd为1
wire wb_acc = wb_cyc_i & wb_stb_i;      // WISHBONE access
wire wb_rd  = wb_acc & !wb_we_i;        // WISHBONE read
access

// 当变量wb_acc为1、wb_rd为1时，表示开始对Flash芯片的读操作。

```

```

// 所以设置输出信号flash_ce、flash_oe都为0，也就是设置为有效
assign flash_ce = !wb_acc;
assign flash_oe = !wb_rd;

// 因为不涉及对Flash芯片的写操作，所以输出信号flash_we始终设置为1
assign flash_we = 1'b1;

assign flash_RST = !wb_RST_i;

always @(posedge wb_clk_i) begin
    if( wb_RST_i == 1'b1 ) begin
        waitstate <= 4'h0;
        wb_ack_o <= 1'b0;
    end else if(wb_acc == 1'b0) begin      // wb_acc为0，表示没
有访问请求
        waitstate <= 4'h0;
        wb_ack_o <= 1'b0;
        wb_dat_o <= 32'h00000000;
    end else if(waitstate == 4'h0) begin // 否则，有访问请
求，开始读操作
        wb_ack_o <= 1'b0;
        if(wb_acc) begin
            waitstate <= waitstate + 4'h1;
        end
        // 给出要读取的第一个字节的地址
        flash_addr_o <=
{10'b0000000000,wb_addr_i[21:2],2'b00};

```

```
end else begin
    // 每个时钟周期将waitstate的值加1
    waitstate <= waitstate + 4'h1;

    if(waitstate == 4'h3) begin
        // 经过3个时钟周期后，第一个字节读到，保存到wb_dat_o[31:24]
        wb_dat_o[31:24] <= flash_dat_i;
        // 给出要读取的第二个字节的地址
        flash_adr_o <=
        {10'b0000000000, wb_adr_i[21:2], 2'b01};

    end else if(waitstate == 4'h6) begin
        // 再经过3个时钟周期后，第二个字节读到，保存到wb_dat_o[23:16]
        wb_dat_o[23:16] <= flash_dat_i;
        // 给出要读取的第三个字节的地址
        flash_adr_o <=
```

```
{10'b000000000000,wb_adr_i[21:2],2'b10};

end else if(waitstate == 4'h9) begin

// 再经过3个时钟周期后，第三个字节读到，保存到wb_dat_o[15:8]

wb_dat_o[15:8] <= flash_dat_i;

// 给出要读取的第四个字节的地址

flash_adr_o <= wb_adr_i[21:2],2'b11};

{10'b000000000000,wb_adr_i[21:2],2'b11};

end else if(waitstate == 4'hc) begin

// 再经过3个时钟周期后，第四个字节读到，保存到wb_dat_o[7:0]

wb_dat_o[7:0] <= flash_dat_i;

// wb_ack_o赋值为1，作为Wishbone总线操作的响应

wb_ack_o <= 1'b1;

end else if(waitstate == 4'hd) begin

// 经过1个时钟周期后，wb_ack_o赋值为0，Wishbone总线操作
```

结束

```
        wb_ack_o  <= 1'b0;  
        waitstate <= 4'h0;  
    end  
end  
endmodule
```

上述代码就是在有读取Flash的请求时（具体而言就是读指令请求），分四次从Flash中读取出四个字节，组成一条指令。每读取一个字节需要3个时钟周期。另外，因为OpenMIPS是大端模式，所以首先读取到的字节对应的是指令的MSB。

13.6 SDRAM控制器

在本章建立的小型SOPC中还具有SDRAM控制器，用来读/写SDRAM。SDRAM的读/写速度快于Flash，而且比Flash成本低，所以一般的片上系统都具有较大容量的SDRAM，作为程序的主运行空间。

小型SOPC中使用的SDRAM控制器是一个开源IP核，但是在配置这个IP核的时候会用到一些SDRAM的相关知识，所以本节先对SDRAM进行简单介绍，熟悉SDRAM的读者可以直接跳至13.6.2节。

13.6.1 SDRAM简介

13.6.1.1 SDRAM结构

SDRAM (Synchronous Dynamic Random Access Memory) 是同步动态随机访问存储器，同步是指Memory工作需要同步时钟，内部命令的发送与数据的传输都以它为基准；动态是指存储阵列需要不断地刷新以保证数据不丢失；随机访问是指数据不是线性依次读写，而是可以自由指定地址进行读/写。

SDRAM的内部有存储单元阵列，给出行地址、列地址，就可以选择对应的存储单元。如图13-9所示。

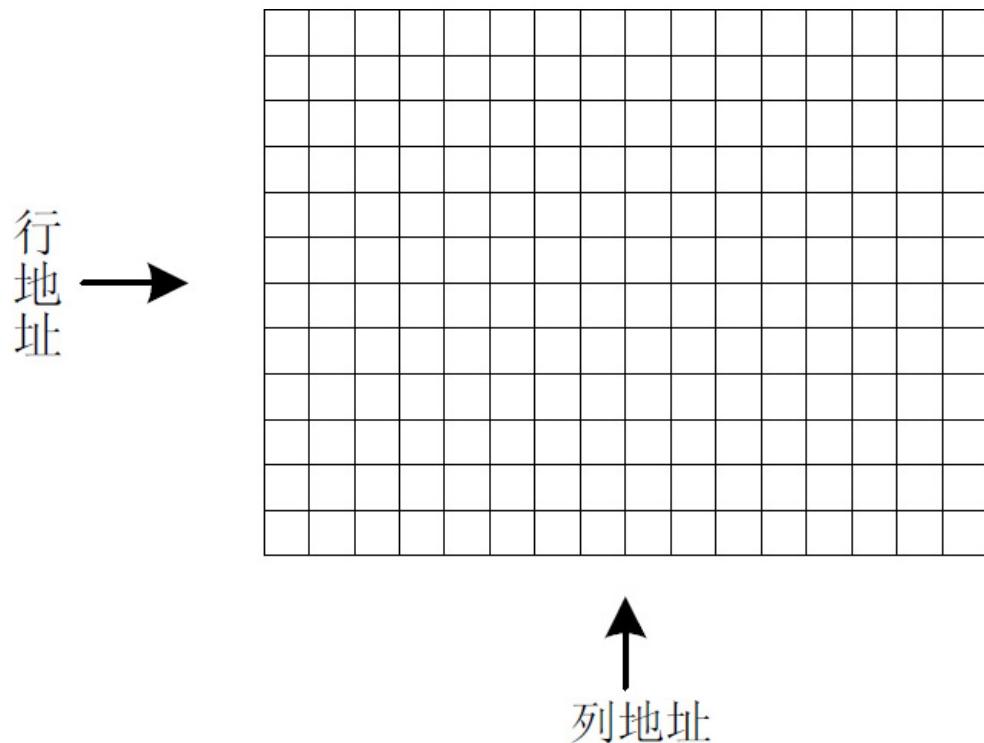


图13-9 SDRAM内部具有存储单元阵列

这样的存储单元阵列称为Bank，在一个SDRAM中往往有多个Bank，寻址的时候需要给出对应Bank的编号。SDRAM的容量就等于“Bank数量*存储单元宽度*地址数”，例如：某型SDRAM有4个Bank，

存储单元宽度是16bit，行地址数是11，列地址数是8，那么该SDRAM的容量就是32Mbit。

在外部接口上，采用了行地址与列地址复用的方式，为此增加了两个接口：行地址选通RAS、列地址选通CAS。当RAS使能时，地址线上的信号是行地址，当CAS使能时，地址线上的信号是列地址。SDRAM的接口一般如表13-14所示。

表13-14 SDRAM的接口

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	ADDR	与具体 SDRAM 有关	输入	地址总线
2	CLK	1	输入	时钟信号
3	CKE	1	输入	时钟使能信号
4	RAS	1	输入	行地址选通信号，低电平有效
5	CS	1	输入	片选信号，低电平有效
6	CAS	1	输入	列地址选通信号，低电平有效
7	WE	1	输入	写操作信号，低电平有效
8	DQM	2	输入	字节选择和输出使能，低电平有效
9	DQ	与具体 SDRAM 有关	双向	数据总线
10	BA	与具体 SDRAM 有关	输入	Bank 选择信号

13.6.1.2 SDRAM的刷新

SDRAM是通过栅极电容存储信息的，由于电容会漏电，所以需要定期进行刷新，以维持原有信息，刷新的方法就是定时重复的对SDRAM进行读出和再写入，以使电容中泄露的电荷得到补充。

13.6.1.3 SDRAM的命令

对SDRAM的访问是通过一系列命令实现的，不同的接口信号组合代表不同的命令，例如：当上一个时钟周期CKE接口为高电平，且本周期CS接口也为高电平，那么表示是“器件不使能”命令。SDRAM主要命令的真值表如表13-15所示。

表13-15 SDRAM主要命令的真值表

命 令	符 号	CKE[n-1]	CKE[n]	CS	RAS	CAS	WE	ADDR[10]	BA
器件不使能	DSEL	H	X	H	X	X	X	X	X
无操作	NOP	H	X	L	H	H	H	X	X
读	READ	H	X	L	H	L	H	L	V
写	WRITE	H	X	L	H	L	L	L	V

续表

命 令	符 号	CKE[n-1]	CKE[n]	CS	RAS	CAS	WE	ADDR[10]	BA
读/自动预取	READAP	H	X	L	H	L	H	H	V
写/自动预取	WRITEAP	H	X	L	H	L	L	H	V
Bank 激活	ACT	H	X	L	L	H	H	V	V
预取激活的 Bank	PRE	H	X	L	L	H	L	L	V
预取所有的 Bank	PALL	H	X	L	L	H	L	H	X
设置模式寄存器	MRS	H	X	L	L	L	L	L	L
自动刷新	CBR	H	H	L	L	L	H	X	X
开始自刷新	SLFRSH	H	L	L	L	L	H	X	X
结束自刷新	SLFRSHX	L	H	H	X	X	X	X	X

注：“H”代表高电平；“L”代表低电平；“V”代表有效数据；“X”代表可以是任何值

13.6.1.4 SDRAM初始化

SDRAM在加电后，必须首先按照预定的方式进行初始化，之后才能正常的工作。初始化需要四个步骤，如图13-10所示。

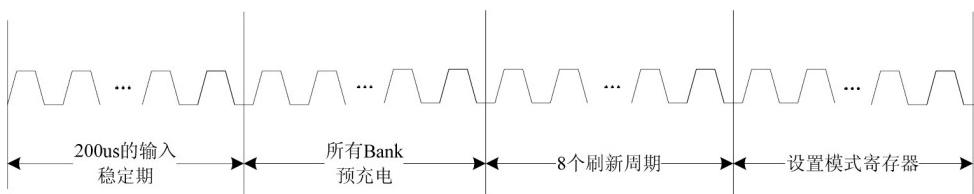


图13-10 SDRAM初始化的四个步骤

第一步：200us的输入稳定期，在这个时间内只有DSEL和NOP命令有效，这个过程实际就是自检过程。

第二步：所有Bank预充电，也就是PALL命令。

第三步：执行8个自动刷新周期，也就是CBR命令。

第四步：设置模式寄存器，也就是MRS命令。模式寄存器的作用将在下一小节讲解。

13.6.1.5 模式寄存器

模式寄存器定义了SDRAM的运行模式，包括CAS延迟（CAS Latency）、突发类型（Burst Type）、突发长度（Burst Length）、工作模式（Operating Mode）、写入突发模式等。模式寄存器的宽度是13bit，分为多个字段，如图13-11所示。读者需要重点关注的是其中三个字段：突发长度、突发类型、CAS延迟。

(1) 突发长度

对SDRAM的读、写操作都是采用突发模式，也就是连续读、写若干个数据（数据的宽度是SDRAM数据接口的宽度）。模式寄存器的第0-2bit定义了突发长度，设定在一次突发传输操作中传输多少数据，可以从1，2，4，8中进行选择，还有的SDRAM支持页模式。

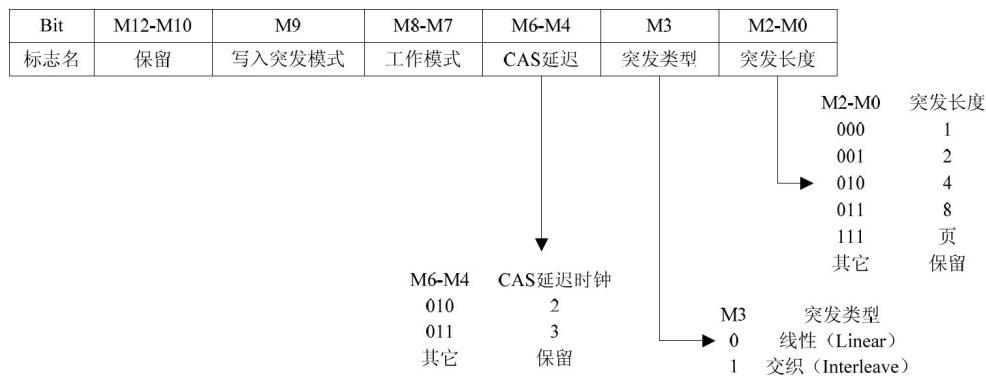


图13-11 模式寄存器的各个字段

(2) 突发类型

模式寄存器的第3bit定义了突发类型，可以有线性 (Linear) 、交织 (Interleave) 两种。不同的突发长度、不同的起始地址、不同的突发类型，会有不同的地址访问顺序，如表13-16所示。

表13-16 突发长度、起始地址、突发类型与地址访问顺序的关系

突发长度	起始地址	突发中的地址访问顺序	
		线性 (Linear)	交织 (Interleave)
2	地址最低位 A0=0	0-1	0-1
2	地址最低位 A0=1	1-0	1-0
4	地址最低两位 A1、A0=00	0-1-2-3	0-1-2-3
4	地址最低两位 A1、A0=01	1-2-3-0	1-0-3-2
4	地址最低两位 A1、A0=10	2-3-0-1	2-3-0-1
4	地址最低两位 A1、A0=11	3-0-1-2	3-2-1-0
8	地址最低三位 A2、A1、A0=000	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
8	地址最低三位 A2、A1、A0=001	1-2-3-4-5-6-7-0	1-0-3-2-5-4-7-6
8	地址最低三位 A2、A1、A0=010	2-3-4-5-6-7-0-1	2-3-0-1-6-7-4-5
8	地址最低三位 A2、A1、A0=011	3-4-5-6-7-0-1-2	3-2-1-0-7-6-5-4
8	地址最低三位 A2、A1、A0=100	4-5-6-7-0-1-2-3	4-5-6-7-0-1-2-3
8	地址最低三位 A2、A1、A0=101	5-6-7-0-1-2-3-4	5-4-7-6-1-0-3-2
8	地址最低三位 A2、A1、A0=110	6-7-0-1-2-3-4-5	6-7-4-5-2-3-0-1
8	地址最低三位 A2、A1、A0=111	7-0-1-2-3-4-5-6	7-6-5-4-3-2-1-0

(3) CAS延迟

模式寄存器的第4-6bit设定CAS延迟。CAS延迟指的是从READ命令发出到第一次数据输出之间的时间，单位是时钟周期。通常设定为2、3个时钟周期。也就是说，如果READ命令在第n个时钟上升沿被触发，那么数据将会在第 $n+ CAS$ 个时钟上升沿开始输出。图13-12是CAS为2的情况，图13-13是CAS为3的情况。

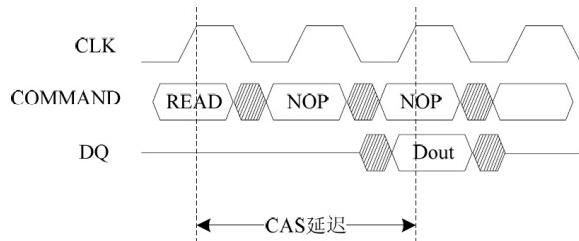


图13-12 CAS为2的情况

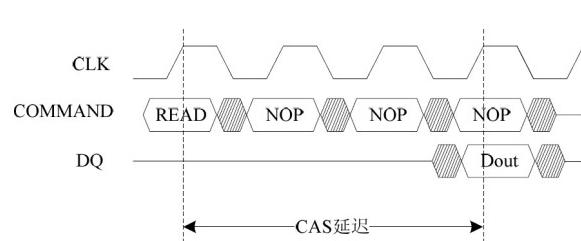


图13-13 CAS为3的情况

13.6.1.6 Bank行激活

SDRAM中有多个Bank，在进行任何读、写操作之前，要先选择进行操作的Bank，然后激活这个Bank中相应的行，通过执行命令ACT来完成这个工作。执行ACT命令时，输入接口BA选择Bank，输入接口A0-Ax选择相应的行。当ACT命令执行完毕之后，需要进行操作的Bank中的对应行就会被打开，这时就可以进行读、写操作了，称为激活。被激活的行会保持激活状态，直到下一次对所在的Bank执行PRE命令。

ACT命令执行完毕之后，还不可以立即进行读、写操作，需要等待一段时间，这段时间称为RAS to CAS Delay，标记为tRCD，一般情况下，tRCD占用2-3个时钟周期。如图13-14所示。

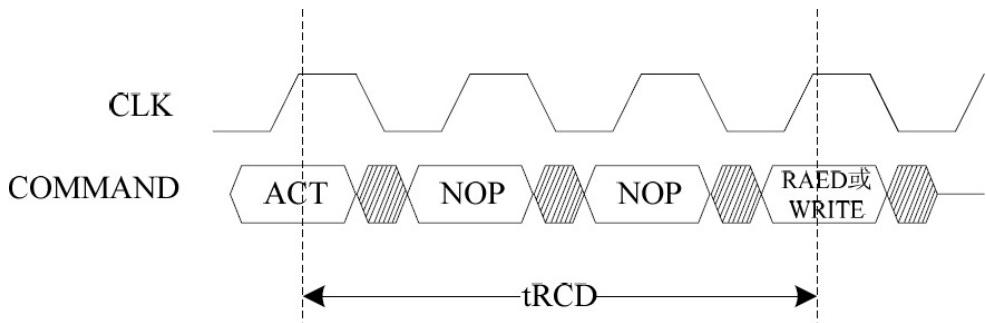


图13-14 ACT命令与读、写命令之间要间隔tRCD时间

13.6.1.7 SDRAM读写

当行地址选定，并且相应的行被激活后，就可以进行读操作了。读操作使用READ或READAP命令，一般是突发读，连续读出若干个数据。第一个数据在经过指定的CAS延时后会出现在数据线上，以后每个时钟都会读出一个新的数据，直到达到设定的突发长度。

写操作也是类似的，使用WRITE或WRITEAP命令，一般是突发写，连续写入若干个数据。第一个要写入的数据与写命令在同一时间给出，以后每个时钟给出下一个要写入的数据，直到达到设定的突发长度。

需要注意的是，由于SDRAM的寻址具有独占性，所以在进行完读、写操作后，如果要对同一Bank的另一行进行寻址，那么要将原来有效（工作）的行关闭，重新发送行/列地址。可以使用预充电命令PRE实现关闭现有工作行、准备打开新行的操作。

13.6.1.8 SDRAM的时间参数

在SDRAM的使用中，有许多时间参数需要注意，在之前的几小节中也提到过一些，此处将主要的时间参数列举如下。

(1) CL (CAS Latency)

CL表示CAS延迟，指的是从READ命令发出到第一次数据输出之间的时间，通常设定为2或3个时钟周期。

(2) tWR (Write Recovery time)

tWR表示写入/矫正时间：在执行写操作时，数据并不是即时地写入存储电容，因为选通三极管与电容的充电必须要有一段时间，所以，数据的真正写入要有一定的周期。为了保证数据的可靠写入，需要留出足够的时间，也就是此处的tWR，一般大于等于1个时钟周期。

(3) tRAS (Active to Precharge Command)

tRAS表示ACT命令与预充电命令之间的时间间隔。预充电命令至少要在行有效命令5个时钟周期后发出，最长间隔视芯片而异，否则工作行的数据有丢失的危险。

(4) tRCD (RAS to CAS Delay)

参考13.6.1.6小节。

(5) tRP (Precharge Command Period)

预充电命令后，需要相隔tRP时间，才可以打开新的行。

(6) tRC (Row Cycle time)

t_{RC} 表示SDRAM行周期时间，它是包括行预充电到激活在内的整个过程所需要的最小时钟周期数。一般而言， $t_{RC} = t_{RAS} + t_{RP}$ 。

(7) tREF (Refresh Period)

t_{REF} 是刷新周期，SDRAM需要定期进行刷新，以维持原有信息，存储体中电容的数据有效保存期的上限是64ms，也就是说每一行刷新的循环周期最多是64ms，刷新速度的计算方法是：行数量/64ms。

13.6.2 SDRAM CONTROLLER IP核

通过上面的学习，读者朋友可能觉得SDRAM很复杂，是的，是很复杂，不过我们并不需要直接操作SDRAM，可以通过SDRAM控制器进行，如图13-3所示，SDRAM控制器位于总线与SDRAM之间，从总线接收访问请求，然后转化为SDRAM的操作命令，将操作结果传递回总线。而SDRAM控制器有开源的IP核可以使用。本章建立的小型SOPC中使用的就是OpenCores站点提供的开源SDRAM控制器——SDRAM CONTROLLER。用户只需要根据自己实际的SDRAM芯片，配置该IP核的一些参数即可。

读者可以在OpenCores站点下载该IP核的源代码，也可以直接在本书附带光盘的Code\Chapter13\sDRAM_controller目录下找到所有源代码。另外，本书附带光盘的Doc目录下提供了该IP核的说明手册。SDRAM CONTROLLER具有以下特点。

- 支持SDRAM的数据总线宽度可以为8、16、32。
- 列地址宽度可配置。
- 支持4个Bank的SDRAM。
- CAS延迟可配置。
- 支持数据掩码，从而实现“部分写”操作（Partial Write Operation）。
- 自动控制刷新。
- 支持所有标准的SDRAM功能。
- 支持Wishbone B版本。

该IP核的接口可以分为两类，一类是Wishbone总线侧的接口，另一类是SDRAM芯片侧的接口，分别如表13-17、表13-18所示。

表13-17 SDRAM CONTROLLER IP核的Wishbone总线接口信号

序号	接 口 名	宽 度 (bit)	输入/输出	作 用
1	wb_clk_i	1	输入	Wishbone 总线时钟信号
2	wb_RST_i	1	输入	Wishbone 总线复位信号
3	wb_eyc_i	1	输入	Wishbone 总线周期信号
4	wb_addr_i	32	输入	Wishbone 总线输入的地址
5	wb_dat_i	32	输入	Wishbone 总线输入的数据
6	wb_we_i	1	输入	Wishbone 总线写使能信号
7	wb_sel_i	4	输入	Wishbone 总线字节选择信号
8	wb_stb_i	1	输入	Wishbone 总线选通信号
9	wb_dat_o	32	输出	Wishbone 总线输出的数据
10	wb_ack_o	1	输出	Wishbone 总线输出的响应
11	wb_cti_i	3	输入	周期类型识别地址标签，是 Wishbone B3 版本才有的信号

表13-18 SDRAM CONTROLLER IP核的SDRAM接口信号

序号	接口名	宽度 (bit)	输入/输出	作用
1	sdr_cke	1	输出	时钟使能信号
2	sdr_cs_n	1	输出	片选, 低电平有效
3	sdr_ras_n	1	输出	行地址选通信号, 低电平有效
4	sdr_cas_n	1	输出	列地址选通信号, 低电平有效
5	sdr_we_n	1	输出	写操作信号, 低电平有效
6	sdr_dqm	可配置	输出	字节选择和输出使能, 低电平有效, 默认宽度为 2
7	sdr_ba	2	输出	Bank 选择信号
8	sdr_addr	13	输出	地址总线
9	sdr_dq	可配置	双向	数据总线, 默认宽度是 16
10	sdr_init_done	1	输出	SDRAM 初始化完毕信号, 为 1 表示初始化完毕

表13-18中的信号与表13-14中SDRAM芯片的接口是一一对应的，在使用的时候，只需要将对应的接口连接起来即可。需要说明两点。

(1) 数据总线的宽度是可配置的，默认是16，还可以配置为8、32。

(2) sdr_init_done是一个比较特别的信号，用来指出SDRAM是否初始化完毕，也就是说，该SDRAM CONTROLLER IP核具有SDRAM初始化功能，用户不需要按照13.6.1.4小节中的说明自己初始化SDRAM，只需要等待sdr_init_done变为1即可，此时的SDRAM就已经初始化完毕了。

除了上面的接口外，要使用该IP核，还需要配置一些参数，如表13-19所示。

表13-19 SDRAM CONTROLLER IP核的一些配置参数

序号	参数名	宽度 (bit)	作用
1	cfg_sdr_width	2	SDRAM的数据总线宽度

			据总线宽度： 00——32位SDRAM, 01——16位SDRAM 1x——8位SDRAM
2	cfg_sdr_en	1	SDRAM控制器使能信号
3	cfg_colbits	2	列地址宽度： 00——8bit 01——9bit 10——10bit 11——11bit
4	cfg_sdr_mode_reg	13	模式寄存器
5	cfg_sdr_tras_d	4	时间tRAS的值，单位是时钟周期
6	cfg_sdr_trp_d	4	时间tRP的值，单位是时钟周期
7	cfg_sdr_trcd_d	4	时间tRCD的值，单位是时钟周期
8	cfg_sdr_cas	3	时间CL的

			值，单位是时钟周期
9	cfg_sdr_trcar_d	4	时间tRC的值，单位是时钟周期
10	cfg_sdr_twrd	4	时间tWR的值，单位是时钟周期
11	cfg_sdr_rfsh	12	自动刷新命令之间的时间间隔，单位是时钟周期
12	cfg_sdr_rfmax	3	每次刷新的最大行数
13	cfg_req_depth	2	请求缓存的数量

从表13-19可知，为了使用SDRAM CONTROLLER IP核，需要配置很多参数，这些参数都是与具体的SDRAM芯片有关，可以查询对应的芯片手册。本章设计的小型SOPC将在DE2开发平台上运行，在DE2上有一个8MB的SDRAM，型号是Zentel公司的A3V64S40ETP-G6，数据宽度是16位，有4个Bank，针对该型SDRAM，设置SDRAM CONTROLLER IP核的配置参数如表13-20所示。

表13-20 针对A3V64S40ETP-G6的配置参数

序号	参数名	宽度 (bit)	配置值
1	cfg_sdr_width	2	2'b01
2	cfg_sdr_en	1	1'b1
3	cfg_colbits	2	2'b00
4	cfg_sdr_mode_reg	13	13'b0000000110001
5	cfg_sdr_tras_d	4	4'b1000
6	cfg_sdr_trp_d	4	4'b0010
7	cfg_sdr_trcd_d	4	4'b0010
8	cfg_sdr_cas	3	3'b100
9	cfg_sdr_trcar_d	4	4'b1010
10	cfg_sdr_twr_d	4	4'b0010
11	cfg_sdr_rfsh	12	12'b011010011000
12	cfg_sdr_rfmax	3	3'b100
13	cfg_req_depth	2	2'b11

有以下几点说明。

(1) 模式寄存器配置为13'b0000000110001，表示CAS延迟为3个时钟周期，突发长度为2（一次读出16bit，2次正好32bit），突发模式是线性（Linear）。

(2) 参数cfg_sdr_cas是CAS延迟的值，应该等于模式寄存器中配置的值，但是笔者在实验的过程中发现该值不能等于模式寄存器中配置的值，而是要比后者的值大1，所以此处设置为3'b100。

(3) A3V64S40ETP-G6芯片的每个Bank有4096行，此处设置每次刷新的最大行数cfg_sdr_rfmax为4，所以在64ms内要求有1024次刷新，每次刷新之间的时间间隔是(64/1024)ms。另外，小型SOPC计划使用DE2开发平台上提供的27MHz时钟源作为工作时钟，(64/1024)ms即大约1688个时钟周期，所以设置cfg_sdr_rfsh为12'b011010011000。

13.7 实现基于实践版OpenMIPS的小型SOPC

现在，OpenMIPS处理器、Wishbone总线互联矩阵、GPIO模块、UART控制器、Flash控制器、SDRAM控制器都已经准备好了，要实现 在本章一开始设计的小型SOPC，只需要将这些模块连接起来即可，为此，修改openmips_min_sopc.v文件如下，源文件位于本书附带光盘中Code\Chapter13目录下。

```
module openmips_min_sopc(
    input wire      clk,
    input  wire     rst,
    // 新增UART接口
```

```
input  wire          uart_in,
output wire         uart_out,

// 新增16位输入接口
input  wire[15:0]    gpio_i,

// 新增32位输出接口
output wire[31:0]   gpio_o,

// 新增与外部Flash相连的接口
input  wire[7:0]     flash_data_i,
output wire[21:0]    flash_addr_o,
output wire          flash_we_o,
output wire          flash_RST_o,
output wire          flash_OE_o,
output wire          flash_CE_o,

// 新增与外部SDRAM相连的接口
output wire          sdr_clk_o,
output wire          sdr_CS_n_o,
output wire          sdr_CKE_o,
output wire          sdr_RAS_n_o,
output wire          sdr_CAS_n_o,
output wire          sdr_WE_n_o,
output wire[1:0]     sdr_DQM_o,
output wire[1:0]     sdr_BA_o,
output wire[12:0]    sdr_ADDR_o,
```

```
    inout  wire[15:0]      sdr_dq_io

);

wire[7:0]    int;
wire          timer_int;
wire          gpio_int;
wire          uart_int;
wire[31:0]   gpio_i_temp;

wire[31:0]   m0_data_i;
wire[31:0]   m0_data_o;
wire[31:0]   m0_addr_i;
wire[3:0]    m0_sel_i;
wire          m0_we_i;
wire          m0_cyc_i;
wire          m0_stb_i;
wire          m0_ack_o;

wire[31:0]   m1_data_i;
wire[31:0]   m1_data_o;
wire[31:0]   m1_addr_i;
wire[3:0]    m1_sel_i;
wire          m1_we_i;
wire          m1_cyc_i;
wire          m1_stb_i;
wire          m1_ack_o;
```

```
wire[31:0] s0_data_i;
wire[31:0] s0_data_o;
wire[31:0] s0_addr_o;
wire[3:0]   s0_sel_o;
wire        s0_we_o;
wire        s0_cyc_o;
wire        s0_stb_o;
wire        s0_ack_i;

wire[31:0] s1_data_i;
wire[31:0] s1_data_o;
wire[31:0] s1_addr_o;
wire[3:0]   s1_sel_o;
wire        s1_we_o;
wire        s1_cyc_o;
wire        s1_stb_o;
wire        s1_ack_i;

wire[31:0] s2_data_i;
wire[31:0] s2_data_o;
wire[31:0] s2_addr_o;
wire[3:0]   s2_sel_o;
wire        s2_we_o;
wire        s2_cyc_o;
wire        s2_stb_o;
wire        s2_ack_i;
```

```

wire[31:0] s3_data_i;
wire[31:0] s3_data_o;
wire[31:0] s3_addr_o;
wire[3:0] s3_sel_o;
wire s3_we_o;
wire s3_cyc_o;
wire s3_stb_o;
wire s3_ack_i;

wire sdr_clk_o = clk;

// ****
**          第一段：例化实践版OpenMIPS处理器          ****
***** /



openmips openmips0(
    .clk(clk),
    .rst(rst),
    // 指令Wishbone总线接口连接到Wishbone总线互联矩阵的主设备接口1
    .iwishbone_data_i(m1_data_o),
    .iwishbone_ack_i(m1_ack_o),

```

```

        .iwishbone_addr_o(m1_addr_i),
. iwishbone_data_o(m1_data_i),
        .iwishbone_we_o(m1_we_i),
. iwishbone_sel_o(m1_sel_i),
        .iwishbone_stb_o(m1_stb_i),
. iwishbone_cyc_o(m1_cyc_i),

        .int_i(int), // 在下面会说明信号int的含义

        // 数据Wishbone总线接口连接到Wishbone总线互联矩阵的主设备接口0
        .dwishbone_data_i(m0_data_o),
. dwishbone_ack_i(m0_ack_o),
        .dwishbone_addr_o(m0_addr_i),
. dwishbone_data_o(m0_data_i),
        .dwishbone_we_o(m0_we_i),
. dwishbone_sel_o(m0_sel_i),
        .dwishbone_stb_o(m0_stb_i),
. dwishbone_cyc_o(m0_cyc_i),

        .timer_int_o(timer_int)

);

// OpenMIPS处理器的中断输入，此处有时钟中断、UART中断、GPIO中断
assign int = {3'b000, gpio_int, uart_int, timer_int};

/*********************
```

```

***          第二段：例化GPIO          ***

***** ****
** /



gpio_top gpio_top0(


    // GPIO连接到Wishbone总线互联矩阵的从设备接口2

    .wb_clk_i(clk),           .wb_rst_i(rst),
    .wb_cyc_i(s2_cyc_o),      .wb_addr_i(s2_addr_o[7:0]),
    .wb_dat_i(s2_data_o),     .wb_sel_i(s2_sel_o),
    .wb_we_i(s2_we_o),        .wb_stb_i(s2_stb_o),
    .wb_dat_o(s2_data_i),     .wb_ack_o(s2_ack_i),
    .wb_err_o(),


    .wb_inta_o(gpio_int),
    .ext_pad_i(gpio_i_temp),
    .ext_pad_o(gpio_o),        // 连接到32位输出接口
    .ext_padoe_o()


);

// 将SDRAM控制器的输出sdram_init_done也作为GPIO的一个输入
assign gpio_i_temp = {15'h0000, sdram_init_done, gpio_i};

***** ****
** /



***** ****          第三段：例化Flash控制器          ****

```

```

***** ****
** /
flash_top flash_top0(
    // Flash控制器连接到Wishbone总线互联矩阵的从设备接口3
    .wb_clk_i(clk),           .wb_rst_i(rst),
    .wb_adr_i(s3_addr_o),     .wb_dat_o(s3_data_i),
    .wb_dat_i(s3_data_o),     .wb_sel_i(s3_sel_o),
    .wb_we_i(s3_we_o),        .wb_stb_i(s3_stb_o),
    .wb_cyc_i(s3_cyc_o),      .wb_ack_o(s3_ack_i),

    // 与小型SOPC外部接口相连，对外连接Flash芯片
    .flash_adr_o(flash_addr_o),
    .flash_dat_i(flash_data_i),
    .flash_RST(flash_RST_o),   .flash_OE(flash_OE_o),
    .flash_CE(flash_CE_o),     .flash_WE(flash_WE_o)
);

/ ****
**
***** ****       第四段：例化UART控制器       ****
***** ****
** /
uart_top uart_top0(

```

```

// UART控制器连接到Wishbone总线互联矩阵的从设备接口1

.wb_clk_i(clk),           .wb_rst_i(rst),
.wb_addr_i(s1_addr_o[4:0]), .wb_dat_i(s1_data_o),
.wb_dat_o(s1_data_i),      .wb_we_i(s1_we_o),
.wb_stb_i(s1_stb_o),       .wb_cyc_i(s1_cyc_o),
.wb_ack_o(s1_ack_i),       .wb_sel_i(s1_sel_o),

// 串口中断

.int_o(uart_int),
```

// 连接UART接口

```

.stx_pad_o(uart_out),      .srx_pad_i(uart_in),
.cts_pad_i(1'b0),          .dsr_pad_i(1'b0),
.ri_pad_i(1'b0),           .dcd_pad_i(1'b0),
.rts_pad_o(),              .dtr_pad_o()

);
```

```

*****
**                                     第五段：例化SDRAM控制器
*****
```

```

sdrc_top sdrc_top0(
```

```

// SDRAM控制器连接到Wishbone总线互联矩阵的从设备接口0

.wb_rst_i(rst),           .wb_clk_i(clk),
```

```

.wb_stb_i(s0_stb_o),           .wb_ack_o(s0_ack_i),
.wb_addr_i({s0_addr_o[25:2],2'b00}), 
.wb_we_i(s0_we_o),             .wb_dat_i(s0_data_o),
.wb_sel_i(s0_sel_o),           .wb_dat_o(s0_data_i),
.wb_cyc_i(s0_cyc_o),           .wb_cti_i(3'b000),

// 与小型SOPC外部接口相连，对外连接SDRAM
.sdram_clk(clk),              .sdram_resetn(~rst),
.sdr_cs_n(sdr_cs_n_o),         .sdr_cke(sdr_cke_o),
.sdr_ras_n(sdr_ras_n_o),       .sdr_cas_n(sdr_cas_n_o),
.sdr_we_n(sdr_we_n_o),         .sdr_dqm(sdr_dqm_o),
.sdr_ba(sdr_ba_o),             .sdr_addr(sdr_addr_o),
.sdr_dq(sdr_dq_io),          

// SDRAM控制器的一些配置，参考表13-20
.cfg_sdr_width(2'b01),         .cfg_colbits(2'b00),
.cfg_req_depth(2'b11),          .cfg_sdr_en(1'b1),
.cfg_sdr_mode_reg(13'b00000000110001),
.cfg_sdr_tras_d(4'b1000),        .cfg_sdr_trp_d(4'b0010),
.cfg_sdr_trcd_d(4'b0010),        .cfg_sdr_cas(3'b100),
.cfg_sdr_trcar_d(4'b1010),       .cfg_sdr_twr_d(4'b0010),
.cfg_sdr_rfsh(12'b011010011000),
.cfg_sdr_rfmax(3'b100)

// SDRAM初始化完毕信号，通过GPIO的输入接口传给处理器
.sdr_init_done(sdram_init_done),
);


```

```

/*****
**
*****          第六段：例化WB_CONMAX          *****
*****
*/
wb_conmax_top wb_conmax_top0(
    .clk_i(clk),           .rst_i(rst),
    // 主设备接口0，连接到OpenMIPS处理器的数据Wishbone总线接口
    .m0_data_i(m0_data_i), .m0_data_o(m0_data_o),
    .m0_addr_i(m0_addr_i), .m0_sel_i(m0_sel_i),
    .m0_we_i(m0_we_i),     .m0_cyc_i(m0_cyc_i),
    .m0_stb_i(m0_stb_i),   .m0_ack_o(m0_ack_o),
    // 主设备接口1，连接到OpenMIPS处理器的指令Wishbone总线接口
    .m1_data_i(m1_data_i), .m1_data_o(m1_data_o),
    .m1_addr_i(m1_addr_i), .m1_sel_i(m1_sel_i),
    .m1_we_i(m1_we_i),     .m1_cyc_i(m1_cyc_i),
    .m1_stb_i(m1_stb_i),   .m1_ack_o(m1_ack_o),
    // 主设备接口2
    .m2_data_i(`ZeroWord), .m2_data_o(),
    .m2_addr_i(`ZeroWord), .m2_sel_i(4'b0000),
    .m2_we_i(1'b0),        .m2_cyc_i(1'b0),

```

```
.m2_stb_i(1'b0), .m2_ack_o(),
.m2_err_o(), .m2_rty_o(),

// 主设备接口3
.m3_data_i(`ZeroWord), .m3_data_o(),
.m3_addr_i(`ZeroWord), .m3_sel_i(4'b0000),
.m3_we_i(1'b0), .m3_cyc_i(1'b0),
.m3_stb_i(1'b0), .m3_ack_o(),
.m3_err_o(), .m3_rty_o(),

// 主设备接口4
.m4_data_i(`ZeroWord), .m4_data_o(),
.m4_addr_i(`ZeroWord), .m4_sel_i(4'b0000),
.m4_we_i(1'b0), .m4_cyc_i(1'b0),
.m4_stb_i(1'b0), .m4_ack_o(),
.m4_err_o(), .m4_rty_o(),

// 主设备接口5
.m5_data_i(`ZeroWord), .m5_data_o(),
.m5_addr_i(`ZeroWord), .m5_sel_i(4'b0000),
.m5_we_i(1'b0), .m5_cyc_i(1'b0),
.m5_stb_i(1'b0), .m5_ack_o(),
.m5_err_o(), .m5_rty_o(),

// 主设备接口6
.m6_data_i(`ZeroWord), .m6_data_o(),
.m6_addr_i(`ZeroWord), .m6_sel_i(4'b0000),
```

```

.m6_we_i(1'b0), .m6_cyc_i(1'b0),
.m6_stb_i(1'b0), .m6_ack_o(),
.m6_err_o(), .m6_rty_o(),

// 主设备接口7
.m7_data_i(`ZeroWord), .m7_data_o(),
.m7_addr_i(`ZeroWord), .m7_sel_i(4'b0000),
.m7_we_i(1'b0), .m7_cyc_i(1'b0),
.m7_stb_i(1'b0), .m7_ack_o(),
.m7_err_o(), .m7_rty_o(),

// 从设备接口0, 连接到SDRAM控制器
.s0_data_i(s0_data_i), .s0_data_o(s0_data_o),
.s0_addr_o(s0_addr_o), .s0_sel_o(s0_sel_o),
.s0_we_o(s0_we_o), .s0_cyc_o(s0_cyc_o),
.s0_stb_o(s0_stb_o), .s0_ack_i(s0_ack_i),
.s0_err_i(1'b0), .s0_rty_i(1'b0),

// 从设备接口1, 连接到UART控制器
.s1_data_i(s1_data_i), .s1_data_o(s1_data_o),
.s1_addr_o(s1_addr_o), .s1_sel_o(s1_sel_o),
.s1_we_o(s1_we_o), .s1_cyc_o(s1_cyc_o),
.s1_stb_o(s1_stb_o), .s1_ack_i(s1_ack_i),
.s1_err_i(1'b0), .s1_rty_i(1'b0),

// 从设备接口2, 连接到GPIO
.s2_data_i(s2_data_i), .s2_data_o(s2_data_o),

```

```
.s2_addr_o(s2_addr_o), .s2_sel_o(s2_sel_o),
.s2_we_o(s2_we_o), .s2_cyc_o(s2_cyc_o),
.s2_stb_o(s2_stb_o), .s2_ack_i(s2_ack_i),
.s2_err_i(1'b0), .s2_rty_i(1'b0),
```

// 从设备接口3，连接到Flash控制器

```
.s3_data_i(s3_data_i), .s3_data_o(s3_data_o),
.s3_addr_o(s3_addr_o), .s3_sel_o(s3_sel_o),
.s3_we_o(s3_we_o), .s3_cyc_o(s3_cyc_o),
.s3_stb_o(s3_stb_o), .s3_ack_i(s3_ack_i),
.s3_err_i(1'b0), .s3_rty_i(1'b0),
```

// 从设备接口4

```
.s4_data_i(), .s4_data_o(),
.s4_addr_o(), .s4_sel_o(),
.s4_we_o(), .s4_cyc_o(),
.s4_stb_o(), .s4_ack_i(1'b0),
.s4_err_i(1'b0), .s4_rty_i(1'b0),
```

// 从设备接口5

```
.s5_data_i(), .s5_data_o(),
.s5_addr_o(), .s5_sel_o(),
.s5_we_o(), .s5_cyc_o(),
.s5_stb_o(), .s5_ack_i(1'b0),
.s5_err_i(1'b0), .s5_rty_i(1'b0),
```

// 从设备接口6

```
.s6_data_i(), .s6_data_o(),
.s6_addr_o(), .s6_sel_o(),
.s6_we_o(), .s6_cyc_o(),
.s6_stb_o(), .s6_ack_i(1'b0),
.s6_err_i(1'b0), .s6_rty_i(1'b0),

// 从设备接口7
.s7_data_i(), .s7_data_o(),
.s7_addr_o(), .s7_sel_o(),
.s7_we_o(), .s7_cyc_o(),
.s7_stb_o(), .s7_ack_i(1'b0),
.s7_err_i(1'b0), .s7_rty_i(1'b0),

// 从设备接口8
.s8_data_i(), .s8_data_o(),
.s8_addr_o(), .s8_sel_o(),
.s8_we_o(), .s8_cyc_o(),
.s8_stb_o(), .s8_ack_i(1'b0),
.s8_err_i(1'b0), .s8_rty_i(1'b0),

// 从设备接口9
.s9_data_i(), .s9_data_o(),
.s9_addr_o(), .s9_sel_o(),
.s9_we_o(), .s9_cyc_o(),
.s9_stb_o(), .s9_ack_i(1'b0),
.s9_err_i(1'b0), .s9_rty_i(1'b0),
```

```
// 从设备接口10
.s10_data_i(), .s10_data_o(),
.s10_addr_o(), .s10_sel_o(),
.s10_we_o(), .s10_cyc_o(),
.s10_stb_o(), .s10_ack_i(1'b0),
.s10_err_i(1'b0), .s10_rty_i(1'b0),

// 从设备接口11
.s11_data_i(), .s11_data_o(),
.s11_addr_o(), .s11_sel_o(),
.s11_we_o(), .s11_cyc_o(),
.s11_stb_o(), .s11_ack_i(1'b0),
.s11_err_i(1'b0), .s11_rty_i(1'b0),

// 从设备接口12
.s12_data_i(), .s12_data_o(),
.s12_addr_o(), .s12_sel_o(),
.s12_we_o(), .s12_cyc_o(),
.s12_stb_o(), .s12_ack_i(1'b0),
.s12_err_i(1'b0), .s12_rty_i(1'b0),

// 从设备接口13
.s13_data_i(), .s13_data_o(),
.s13_addr_o(), .s13_sel_o(),
.s13_we_o(), .s13_cyc_o(),
.s13_stb_o(), .s13_ack_i(1'b0),
.s13_err_i(1'b0), .s13_rty_i(1'b0),
```

```

// 从设备接口14
.s14_data_i(),           .s14_data_o(),
.s14_addr_o(),           .s14_sel_o(),
.s14_we_o(),             .s14_cyc_o(),
.s14_stb_o(),            .s14_ack_i(1'b0),
.s14_err_i(1'b0),         .s14_rty_i(1'b0),

// 从设备接口15
.s15_data_i(),           .s15_data_o(),
.s15_addr_o(),           .s15_sel_o(),
.s15_we_o(),             .s15_cyc_o(),
.s15_stb_o(),            .s15_ack_i(1'b0),
.s15_err_i(1'b0),         .s15_rty_i(1'b0),
);

endmodule

```

上述代码虽然比较长，但是并不复杂，主要是例化各个模块，并按照图13-5所示的方式连接起来，可以分为六段理解。

第一段：例化了实践版OpenMIPS处理器，其中将数据Wishbone总线接口连接到WB_CONMAX的主设备接口0，将指令Wishbone总线接口连接到WB_CONMAX的主设备接口1。另外，中断输入信号不仅包

括时钟中断，还增加了UART中断、GPIO中断，分别来自UART控制器、GPIO模块。

第二段：例化了GPIO模块，将其连接到WB_CONMAX的从设备接口2。另外，GPIO模块的输入信号是gpio_i_temp，其第16bit是SDRAM控制器的输出信号sdram_init_done，这样做的目的是：处理器可以通过读取GPIO的输入值，判断其第16bit是否为1，从而知道SDRAM是否初始化完毕。

第三段：例化了Flash控制器，将其连接到WB_CONMAX的从设备接口3。

第四段：例化了UART控制器，将其连接到WB_CONMAX的从设备接口1。注意，UART控制器的串行接口部分只使用了收、发两个接口，其他接口（例如：RTS、CTS等流控制接口）都没有使用。

第五段：例化了SDRAM控制器，将其连接到WB_CONMAX的从设备接口0。其中wb_cti_i是Wishbone B3版本才添加的信号，此处并不使用，直接设置为3'b000即可。另外，SDRAM控制器的参数配置是参考表13-20设置的，针对的是A3V64S40ETP-G6芯片。读者如果使用其他型号的SDRAM芯片，那么需要根据芯片的实际情况修改此处的参数配置。

第六段：例化了WB_CONMAX。只使用了主设备接口0、1，以及从设备接口0、1、2、3，其余接口都没有使用。

通过上述代码，可以得到本章设计的基于实践版OpenMIPS的小型SOPC的接口如图13-15所示，详细说明参考表13-21。

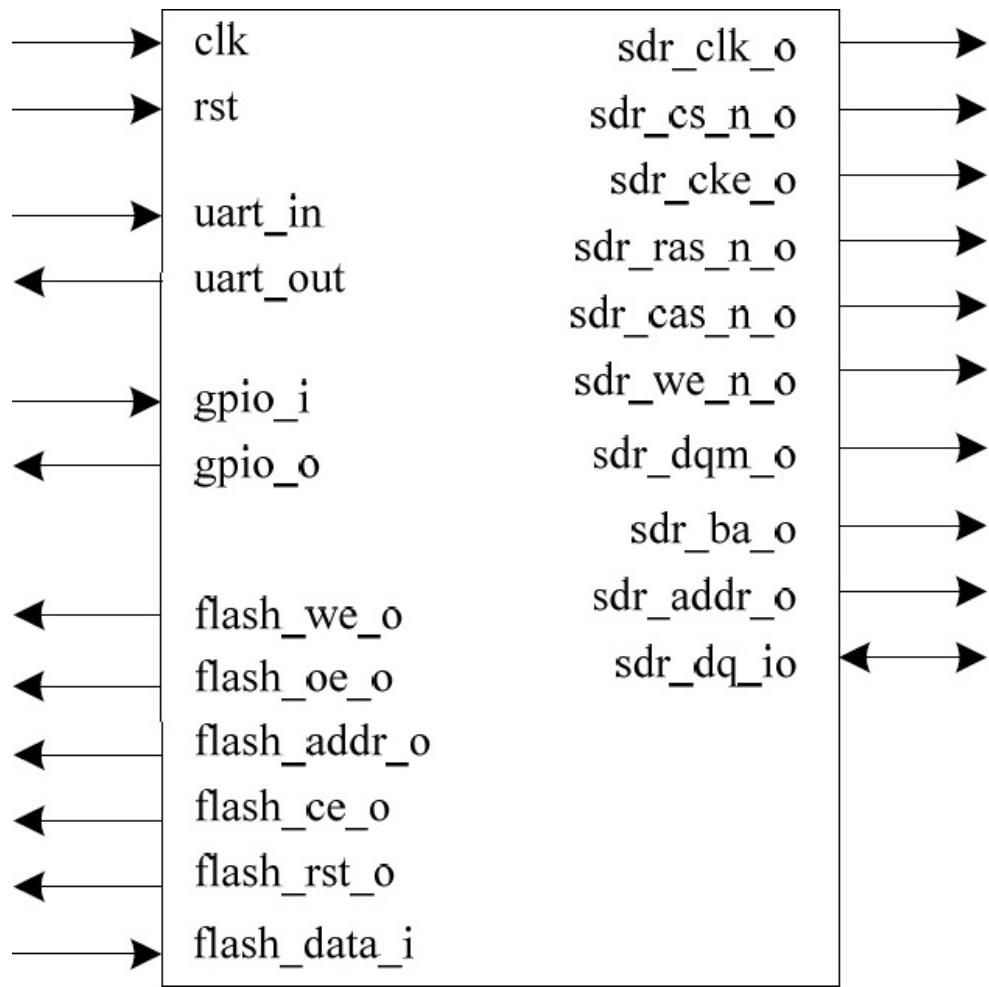


图13-15 基于实践版OpenMIPS的小型SOPC的接口

表13-21 基于实践版OpenMIPS的小型SOPC的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	clk	1	输入	时钟
2	rst	1	输入	复位信号
3	uart_in	1	输入	串口输入信号
4	uart_out	1	输出	串口输出信号
5	gpio_i	16	输入	GPIO 输入信号
6	gpio_o	32	输出	GPIO 输出信号
7	flash_oe_o	1	输出	Flash 输出使能信号, 低电平有效
8	flash_addr_o	22	输出	Flash 地址信号
9	flash_ce_o	32	输出	Flash 片选信号, 低电平有效
10	flash_RST_o	1	输出	Flash 复位信号, 低电平有效
11	flash_we_o	1	输出	Flash 写使能信号, 低电平有效
12	flash_data_i	8	输入	从 Flash 读出的数据

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
13	sdr_clk_o	1	输出	SDRAM 时钟信号
14	sdr_es_n_o	1	输出	SDRAM 片选信号, 低电平有效
15	sdr_cke_o	1	输出	SDRAM 时钟使能信号
16	sdr_ras_n_o	1	输出	SDRAM 行地址选通信号, 低电平有效
17	sdr_cas_n_o	1	输出	SDRAM 列地址选通信号, 低电平有效
18	sdr_we_n_o	1	输出	SDRAM 写操作信号, 低电平有效
19	sdr_dqm_o	2	输出	SDRAM 字节选择和输出使能, 低电平有效
20	sdr_ba_o	2	输出	SDRAM 的 Bank 选择信号
21	sdr_addr_o	13	输出	SDRAM 地址总线
22	sdr_dq_io	16	双向	SDRAM 数据总线

13.8 本章小结

为了验证实践版OpenMIPS处理器是否实现正确，本章搭建了基于实践版OpenMIPS处理器的小型SOPC。该SOPC包括GPIO、UART控制器、Flash控制器、SDRAM控制器等模块，这些模块都具有Wishbone总线接口，与OpenMIPS处理器一起挂接在Wishbone总线上，便于处理器访问。此外，这些模块中的大多数都是采用已有的开源IP核，只有Flash控制器是自行设计的。下一章将在该SOPC之上运行测试程序，进行检验。

第14章 验证实践版OpenMIPS处理器

上一章设计实现了基于实践版OpenMIPS处理器的小型SOPC，本章将把该小型SOPC下载到实际的FPGA芯片，并运行测试程序，通过检验测试程序的执行效果，验证实践版OpenMIPS处理器是否实现正确。

首先简单介绍了要用到的开发平台DE2，由于DE2上的FPGA芯片是Altera公司的CycloneII系列，所以需要使用QuartusII这个软件来建立工程，14.3节给出了QuartusII工程的建立方法，同时说明了小型SOPC的接口与FPGA芯片引脚的对应关系。14.4节说明了测试步骤。14.5节进行了GPIO实验，14.6节进行了UART实验，14.7节是一个综合实验，其模拟了操作系统的加载过程。

有的读者朋友可能会抱怨，自己拥有的开发平台不是DE2，没关系，步骤都是相似的，大家可以参考本章的内容，在自己的开发平台上验证、使用OpenMIPS处理器。

14.1 DE2平台简介

DE2是Altera公司针对大学教学及研究机构推出的FPGA多媒体开发平台，提供了丰富的外设及多媒体特性，其核心FPGA芯片是Altera CycloneII系列的EP2C35F672。开发平台的外观如图14-1所示。

DE2的主要资源列举如下。

- (1) Altera Cyclone II系列FPGA芯片EP2C35F672，内含35000个逻辑单元（LE）。
- (2) 主动串行配置器件EPCS16。
- (3) 板上内置用于编程调试的USB Blaster，支持JTAG模式和AS模式。
- (4) 512KB的SRAM、8MB的SDRAM、4MB的NOR Flash。
- (5) 具有SD卡接口、PS2接口。
- (6) 具有红外IrDA收发器。
- (7) 4个按键、18个拨动开关、8个绿色LED灯、18个红色LED灯，以及8个7段数码管。
- (8) 板载50MHz、27MHz两个晶振可选择作为系统时钟，也可使用外部时钟。
- (9) 24位CD品质音频编解码器，带有麦克风输入插座、线性输入插座和线性输出插座。
- (10) VGA数模转换器，内含3个10位高速DAC。
- (11) 支持NTSC和PAL制式的视频解码器ADV7181B。
- (12) 10/100M以太网控制器DM9000AE及网络接口。

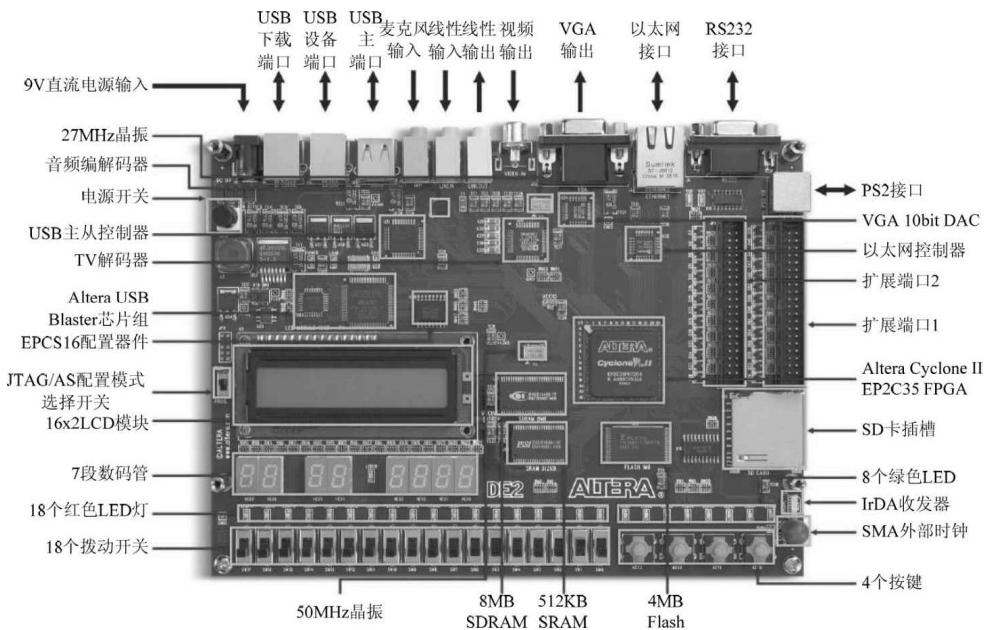


图14-1 DE2开发平台

(13) USB主从控制器及接口。

(14) RS-232收发器及DB9针接口。

(15) 16x2字符的LCD模块。

(16) 两个40针脚的扩展端口。

从上述介绍可知DE2具有丰富的资源，但是我们小型SOPC只是使用到了其中一部分。更加详细的说明可以参考本书附带光盘中Doc目录下的PDF文档DE2_UserManul。

14.2 测试需要的硬件连接

测试需要的硬件连接很简单，如图14-2所示。PC与DE2平台之间只需要两根线，一根连接PC上的USB接口与DE2上的USB Blaster接

口，供下载调试使用。另一跟连接PC上的串口与DE2上的串口，这样小型SOPC就可以通过串口给PC发送数据，PC也可以通过串口给小型SOPC发送数据。

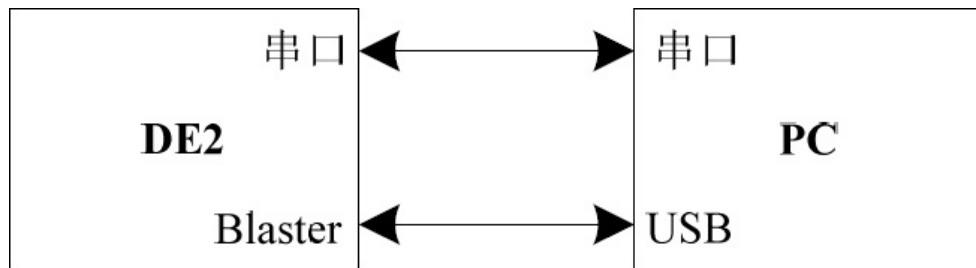


图14-2 测试需要的硬件连接

14.3 QuartusII工程建立

针对Altera公司的FPGA，需要使用QuartusII建立工程，然后编译得到可以下载到FPGA上的配置文件。QuartusII工程建立步骤如下。

打开QuartusII，选择File->New Project Wizard，如图14-3所示。

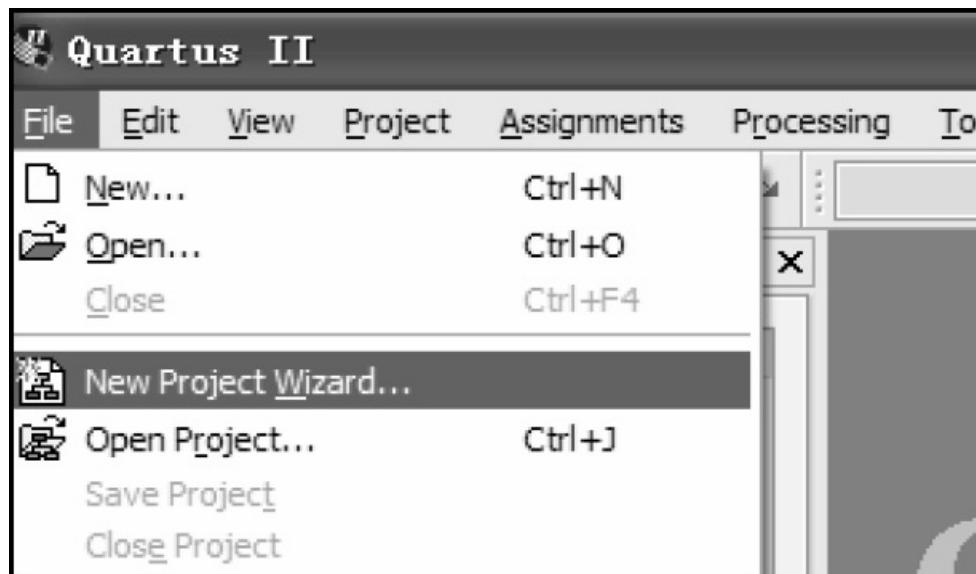


图14-3 使用“新工程向导”

会出现如图14-4所示的对话框，在其中设置工程保存路径、工程名称。其中，工程名称可以设置为openmips_min_sopc。

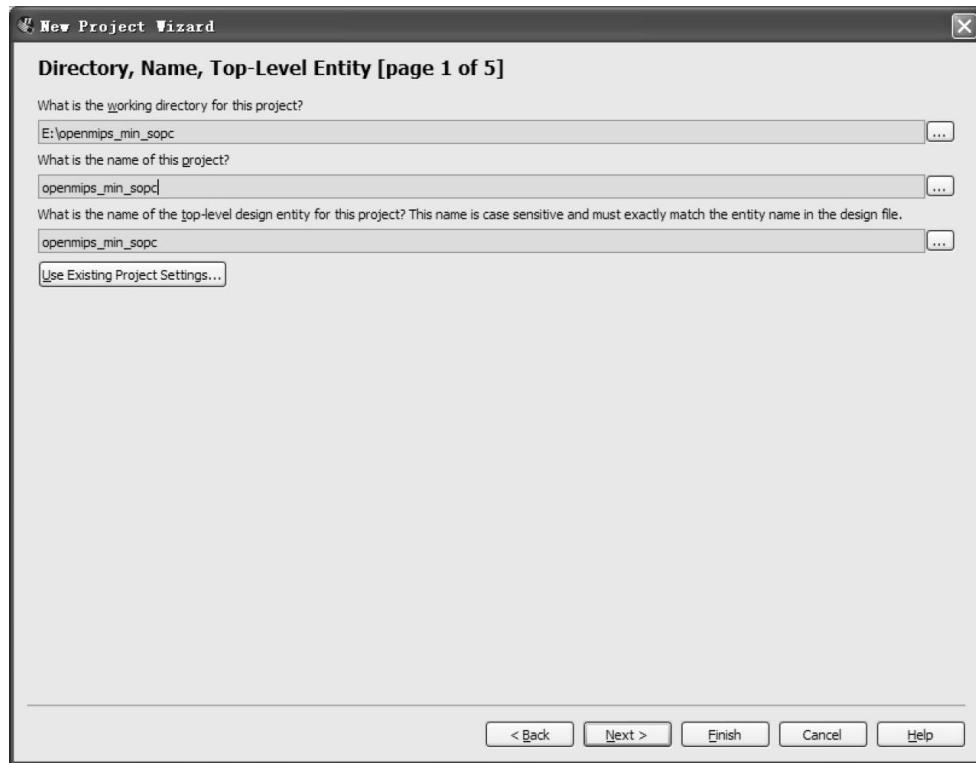


图14-4 设置工程保存路径、工程名称

单击Next按钮，会出现添加文件对话框，如图14-5所示。单击File Name后面的省略号按钮，添加本书附带光盘中Code\Chapter13目录下的所有文件。

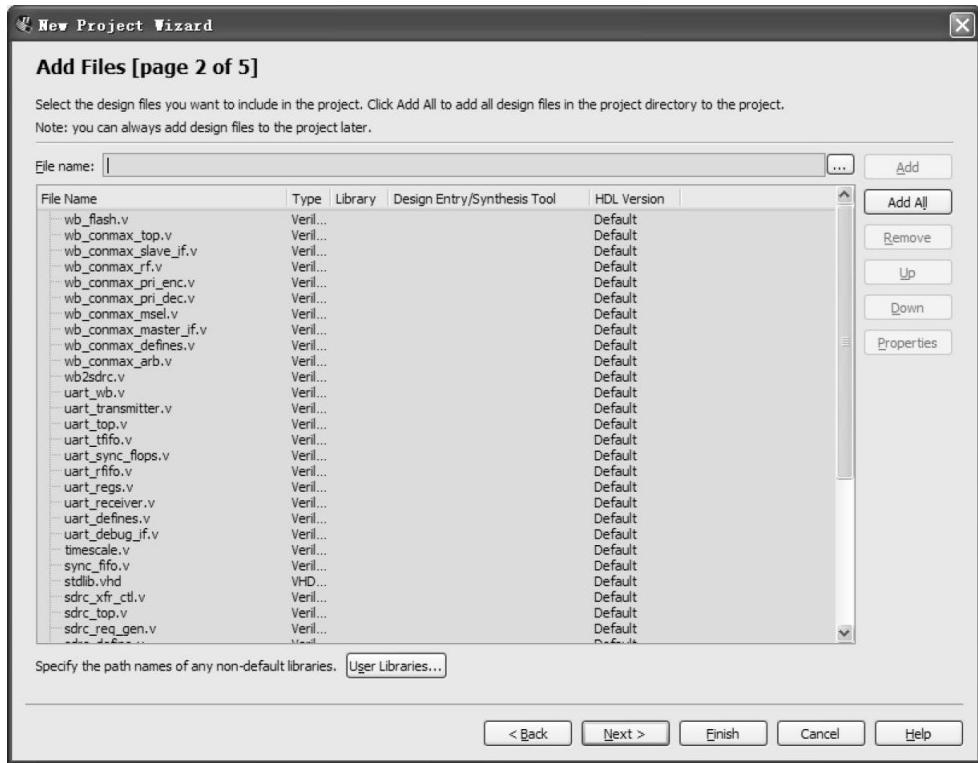


图14-5 添加光盘Code\Chapter13目录下的所有文件

单击Next按钮，会出现器件选择对话框，如图14-6所示，在其中选择目标平台FPGA芯片的型号，针对DE2平台，此处选择CycloneII系列的EP2C35F672C6作为目标器件，

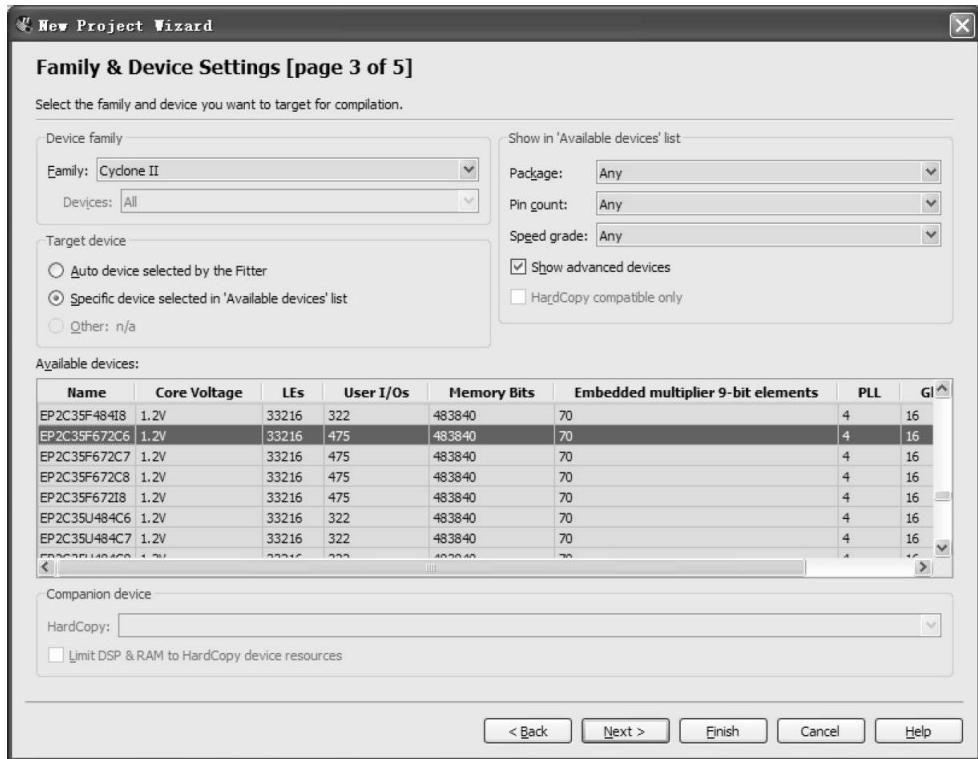


图14-6 选择目标平台上FPGA芯片的型号

然后一直单击Next按钮，在最后一步单击Finish按钮，这样就建立了QuartusII工程。单击工具栏上的Start Compilation按钮，将编译整个工程，如图14-7所示。



图14-7 点击Start Compilation按钮将编译整个工程

编译会持续5~10分钟左右的时间，编译完成之后，可以进行引脚配置了，单击Assignments->Pin Planner菜单选项，如图14-8所示。

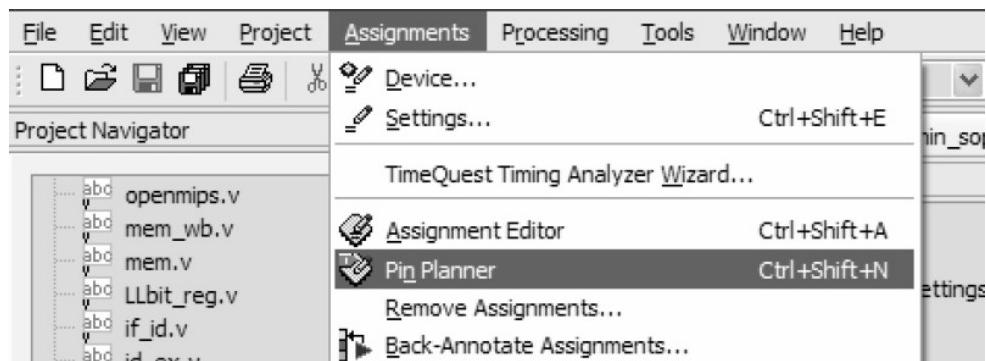


图14-8 点击Pin Planner进行引脚配置

出现引脚配置窗口，如图14-9所示。在其中将小型SOPC的各个接口与FPGA芯片的引脚对应起来。

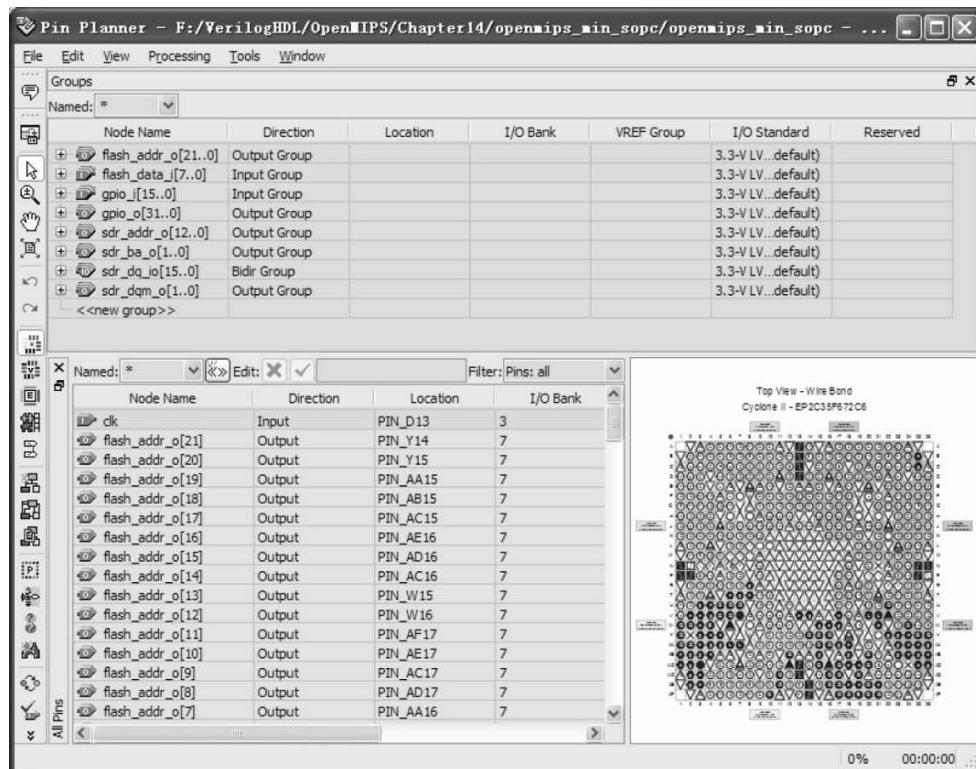


图14-9 引脚配置窗口

从第13章的图13-15可知，小型SOPC的接口有五类：系统接口（复位rst、时钟clk）、GPIO接口（输入gpio_i、输出gpio_o）、

UART接口（接收uart_in、发送uart_out）、Flash接口、SDRAM接口，分别连接DE2平台的如下资源。

- 时钟接口clk连接到DE2上的27MHz时钟源。
- 复位接口rst连接到DE2上的拨动开关SW17，也就是图14-1中，左下角的那个拨动开关。
- GPIO输入接口gpio_i连接到DE2上的16个拨动开关SW0-SW15。
- GPIO输出接口gpio_o连接到DE2上的4个7段数码管。
- UART接口与DE2板上的UART收、发端口一一对应。
- Flash接口与DE2板上的Flash芯片的引脚一一对应。
- SDRAM接口与DE2板上的SDRAM芯片的引脚一一对应。

上述连接如图14-10所示。

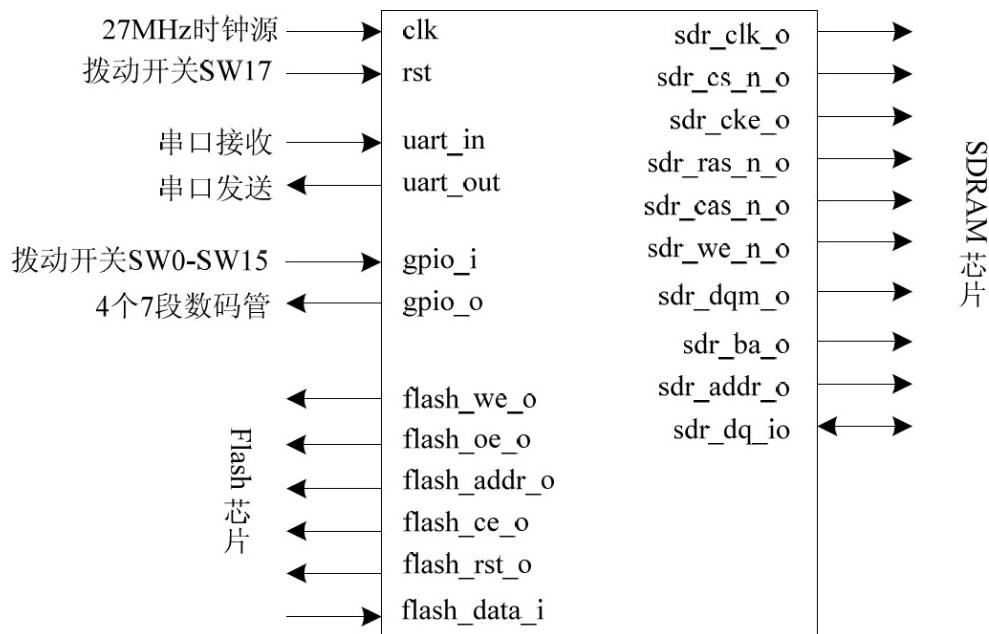


图14-10 小型SOPC与DE2平台资源的连接关系

配置完引脚后，再次编译该QuartusII工程，得到可以下载到FPGA中的配置文件openmips_min_sopc.sof。同时得到编译报告，显示资源占用情况，如图 14-11 所示。在本书附带光盘Code\Chapter14\openmips_min_sopc 目录下提供了完整的QuartusII 工程。

Flow Status	Successful - Tue Apr 15 18:06:48 2014
Quartus II Version	10.1 Build 197 01/19/2011 SP 1 SJ Full Version
Revision Name	openmips_min_sopc
Top-level Entity Name	openmips_min_sopc
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	10,831 / 33,216 (33 %)
Total combinational functions	10,269 / 33,216 (31 %)
Dedicated logic registers	4,363 / 33,216 (13 %)
Total registers	4363
Total pins	125 / 475 (26 %)
Total virtual pins	0
Total memory bits	256 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	0 / 4 (0 %)

图14-11 小型SOPC的资源占用情况

14.4 测试步骤说明

上一节编译得到了可以下载到FPGA中的配置文件openmips_min_sopc.sof，但先别着急下载，因为小型SOPC是从Flash启动的，而此时Flash中并没有测试程序。正确的测试步骤如图14-12所示。

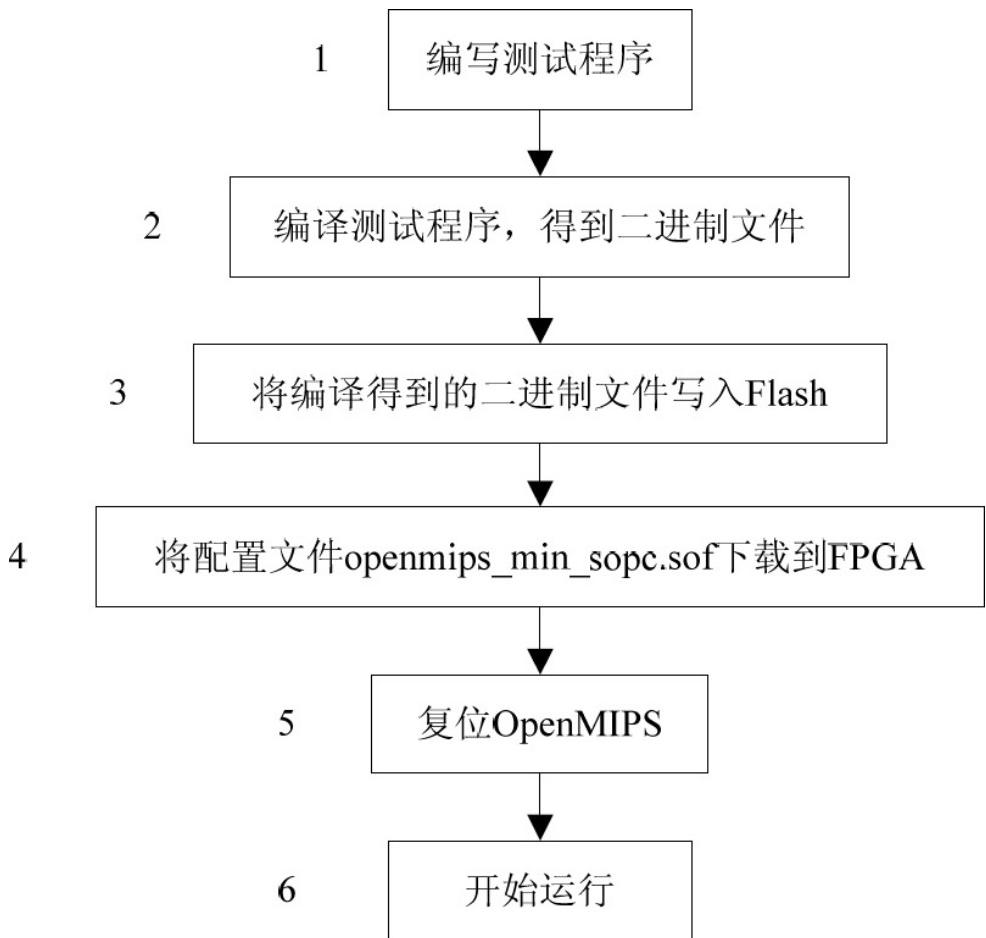


图14-12 正确的测试步骤

其中第3步“将编译得到的二进制文件写入Flash”是通过以下两小步实现的。

(1) 将DE2开发平台附带光盘提供的配置文件DE2_USB_API.sof下载到FPGA。本书附带光盘中的DE2文件夹下也提供了该文件。

(2) 打开DE2开发平台附带光盘提供的程序DE2_Control_Panel.exe，本书附带光盘的DE2文件夹下也提供了该程序。使用该程序首先Erase Flash，然后再将测试程序对应的二进制文件写入Flash。

需要先Erase Flash，然后再写Flash，这是由Flash的特性决定的，Flash可以将一个bit从1变为0，但是不可以从0变为1，所以只有先通过Erase操作，将其全部变为1后，再写入。

图14-12中，第5步的复位操作，实际就是将拨动开关SW17向上拨，使得相应输入为高电平，该开关连接到SOPC的rst接口，所以使得rst的值为1。

图14-12中，第6步的启动操作，实际就是将拨动开关SW17向下拨，使得相应输入为低电平，该开关连接到SOPC的rst接口，所以使得rst的值为0。

以上就是正确的测试步骤，下一节会结合GPIO实验详细说明各个步骤是如何操作的。本书其余的实验就不再详细说明测试步骤，只是给出测试代码、测试结果。

14.5 测试——GPIO实验

14.5.1 测试内容

本测试的主要内容是OpenMIPS处理器控制GPIO的输出。

在14.3节介绍引脚配置的时候提到GPIO输出接口gpio_o与4个7段数码管相连。所以gpio_o的值可以通过7段数码管显示出来。7段数码管的引脚与数码管的对应关系如图14-13所示。如果某一引脚输入电平为低电平，那么对应的数码管就会点亮，比如：如果引脚a输入低电平，那么最上面的数码管就会点亮。

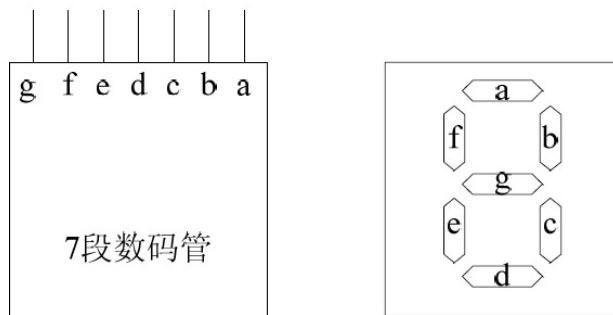


图14-13 7段数码管的引脚与数码管的对应关系

GPIO输出接口gpio_o的宽度为32位，含4个字节，每个字节对应一个数码管，而每个数码管只有7个引脚，所以每个字节都有一位没有使用，此处统一设置为每个字节的最高位没有使用，如图14-14所示。

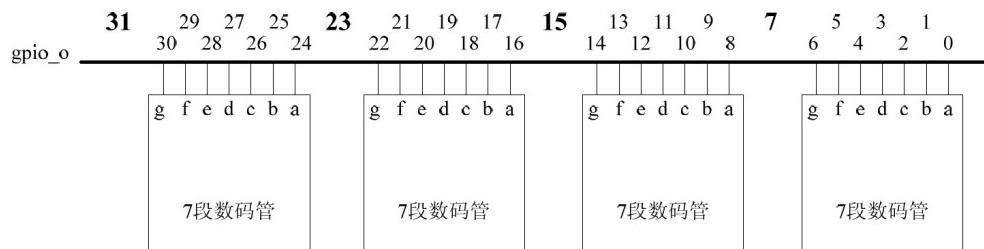


图14-14 gpio_o接口与4个7段数码管的连接示意图

14.5.2 测试程序

测试程序如下，源文件是本书附带光盘中Code\Chapter14\gpio_test目录下的inst_rom.S文件。

```
.org 0x0
.set neat
.set noreorder
.set nomacro
.global _start
```

```
_start:
```

```
#####
```

第一段

```
#####
```

```
lui $1, 0x2000
```

```
ori $1,$1, 0x0008
```

```
lui $2, 0xfffff
```

```
ori $2,$2, 0xfffff
```

```
sw $2, 0x0($1) # 向地址0x20000008写入0xffffffff
```

0x20000008对应GPIO模块的寄存器

```
RGPIO_OE
```

```
#####
```

第二段

```
#####
```

```
lui $1, 0x2000
```

```
ori $1,$1, 0x000c
```

```
lui $2, 0x0000
```

```
ori $2,$2, 0x0000
```

```
sw $2, 0x0($1) # 向地址0x2000000c写入0x00000000
```

0x2000000c对应GPIO模块的寄存器

```
RGPIO_INTE
```

```
#####
```

第三段

```
#####
```

```
lui $1, 0x2000
```

```
ori $1,$1,0x0004  
lui $2,0x4740  
ori $2,$2,0x4106  
sw  $2,0x0($1)          # 向地址0x20000004写入0x47404106  
                                # 0x20000004对应GPIO模块的寄存器  
RGPIO_OUT
```

上述测试程序可以分为三段理解。

第一段：向地址0x20000008写入0xffffffff，GPIO模块连接到Wishbone总线互联矩阵的从设备接口2，所以其地址是从0x20000000开始，通过表13-3可知，偏移为0x8的地址对应的是GPIO_OE寄存器，向该寄存器写入0xffffffff，表示GPIO的32个输出接口都使能。

第二段：向地址0x2000000c写入0x00000000，通过表13-3可知，偏移为0xc的地址对应的是GPIO_INTE寄存器，向该寄存器写入0x00000000，表示中断禁止。

第三段：向地址0x20000004写入0x47404106，通过表13-3可知，偏移为0x4的地址对应的是GPIO_OUT寄存器，向该寄存器写入0x47404106，表示GPIO输出的信号是0x47404106。

如果运行正确，那么4个7段数码的显示应该如图14-15所示，黑色表示数码管被点亮，显示“LOVE”单词。读者可以结合图14-13、图14-14，理解体会为何0x47404106会对应这个显示结果。

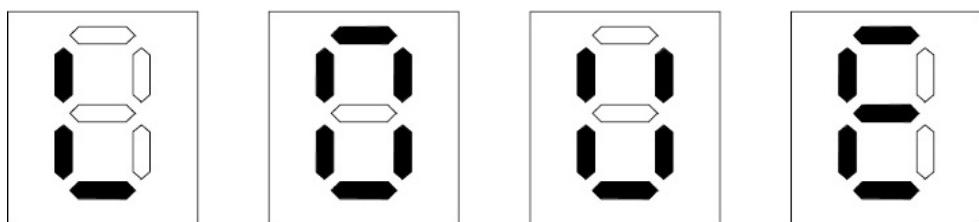


图14-15 4个7段数码管的预期显示效果

14.5.3 编译测试程序

现在需要编译测试程序，将上述inst_rom.S文件，与第4章建立的Bin2Mem.exe、Makefile、ram.ld这三个文件复制到Ubuntu虚拟机中的同一个目录下，打开终端，使用cd命令进入该目录，然后输入make all，即可得到要写入Flash的二进制文件inst_rom.bin。

需要注意一点：在编译前要修改ram.ld文件，将其中的起始地址从0x00000000修改为0x30000000，如下，完整文件位于本书附带光盘中Code\Chapter14\gpio_test目录下。

```
MEMORY
{
    ram      :
ORIGIN = 0x30000000
,
    LENGTH = 0x00000300
}
```

做此修改主要是因为测试程序是在Flash中运行的，而Flash的起始地址就是0x30000000。

14.5.4 将测试程序写入Flash芯片

在14.4节已经说明：将测试程序写入Flash芯片可以分为两小步。

1. 将配置文件DE2_USB_API.sof下载到FPGA。
2. 打开程序 DE2_Control_Panel.exe，使用该程序首先Erase Flash，然后将编译测试程序的得到的二进制文件写入Flash。

具体操作步骤如下。

单击QuartusII工具栏上的Programmer按钮，如图14-16所示。



图14-16 单击工具栏上的Programmer按钮

出现Programmer对话框，如图14-17所示。单击“Change File...”按钮，选择DE2附带光盘提供的DE2_USB_API.sof文件，本书附带光盘的DE2目录下也提供了该文件。选中其中的“Program/Configure”对应的复选框，然后单击“Start”按钮，将该配置文件下载到FPGA。

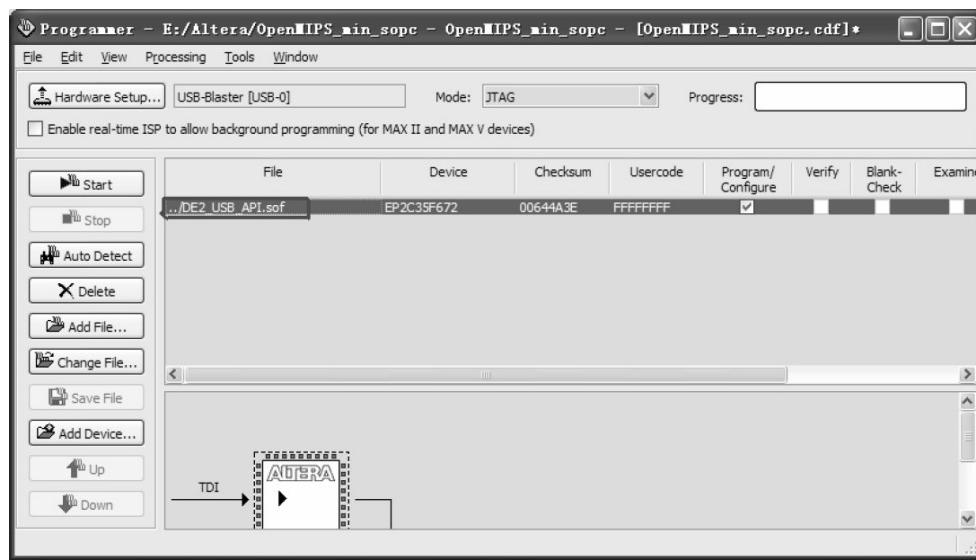


图14-17 Programmer对话框

下载完成后，打开DE2附带光盘或本书附带光盘提供的DE2_Control_Panel.exe程序，选择Open菜单，单击Open USB Port 0选项，如图14-18所示。

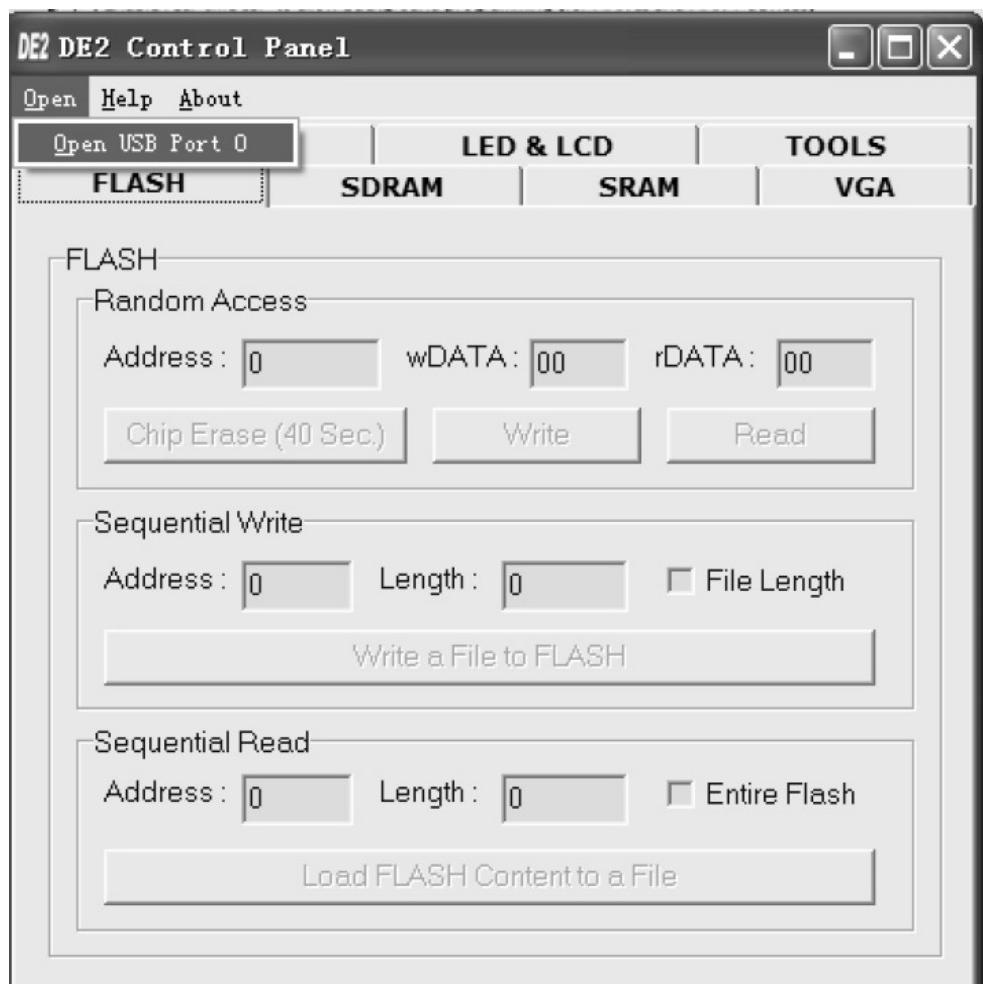


图14-18 Open USB Port

然后选择“FLASH”这个Tab，单击Chip Erase按钮，将Erase Flash，该过程大约需要40秒的时间。如图14-19所示。

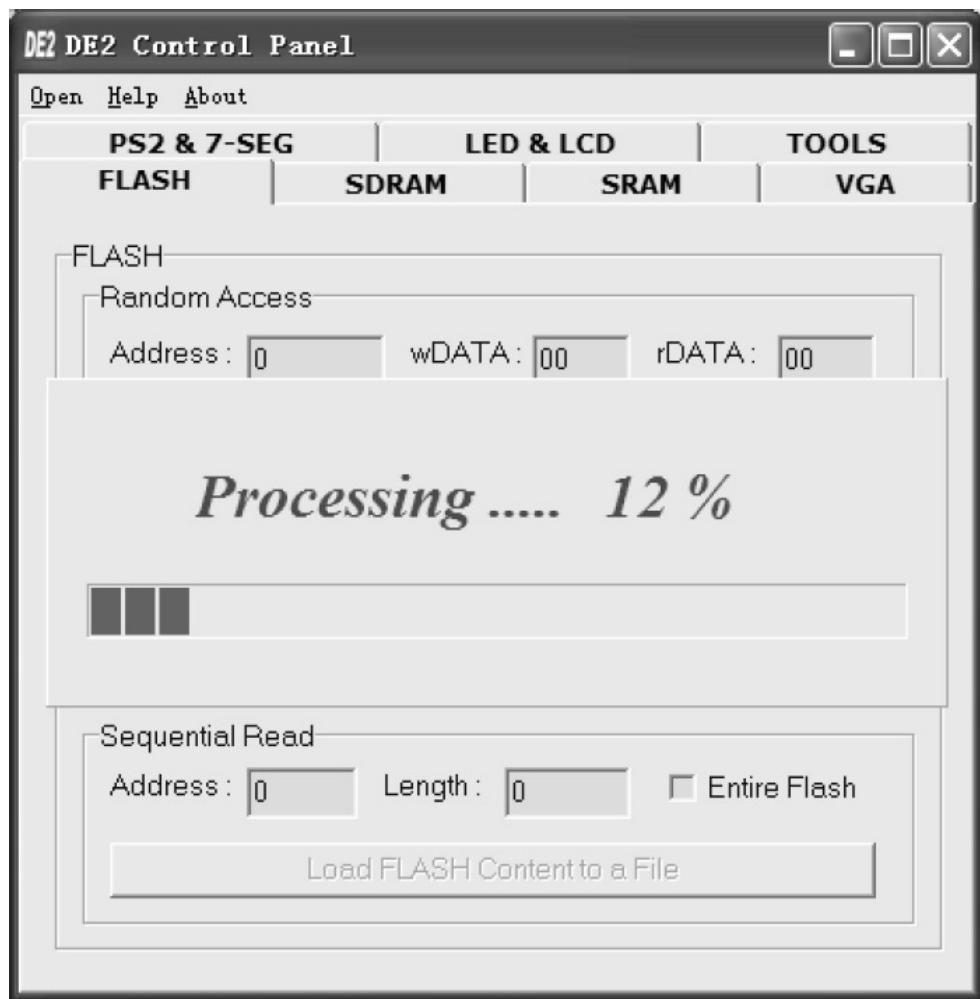


图14-19 Erase Flash的过程

Erase Flash结束后，可以将测试程序写入Flash了，如图14-20所示，选中File Length前面的复选框，单击“Write a File to FLASH”按钮，会出现一个文件选择框，在其中选择之前编译得到的inst_rom.bin文件，就会将该文件写入Flash。

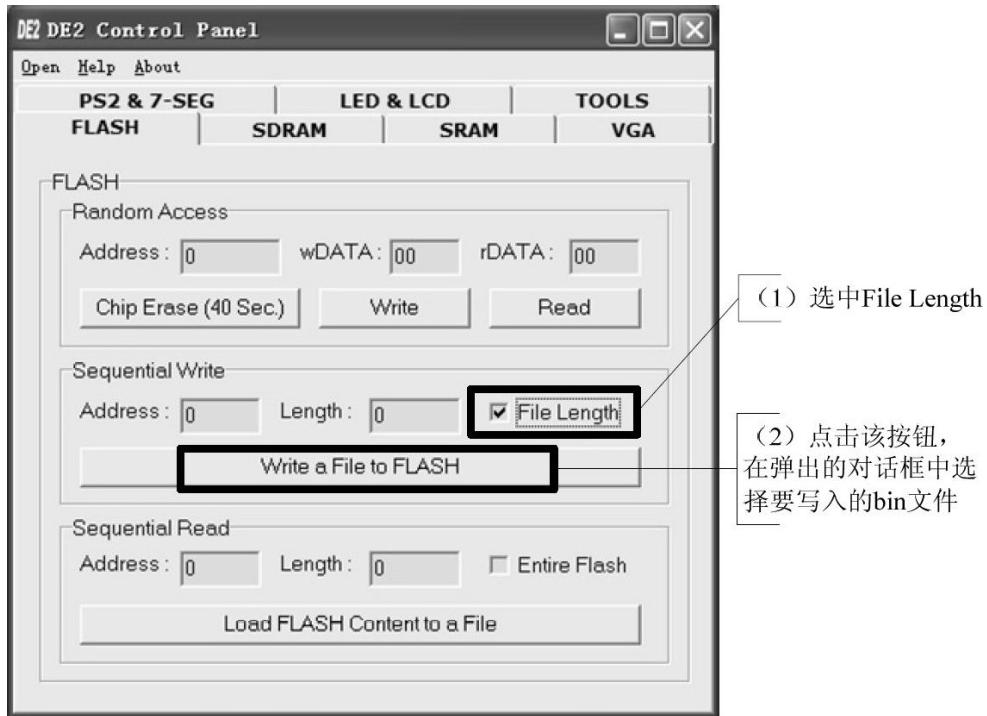


图14-20 将测试程序写入Flash

最后，再次选择Open菜单，单击Close USB Port选项。如图14-21所示。



图14-21 Close USB Port

14.5.5 下载小型SOPC到DE2

到这一步，已经将测试程序写入Flash了，现在，可以向FPGA下载我们早在14.3节就已编译得到的配置文件openmips_min_sopc.sof了。下载过程很简单，再次打开QuartusII的Programmer工具，选择在14.3

节得到的OpenMIPS_min_sopc.sof，然后单击Start按钮，就将该配置文件下载到FPGA了。

14.5.6 测试效果

好了，小型SOPC已经下载到FPGA了，测试程序也已写入Flash了，只需要复位一下SOPC，然后就可以运行了。让我们拨动开关SW17，先向上拨一下，再向下拨一下。DE2上的7段数码管会呈现如图14-22所示的效果。这证明我们的实践版OpenMIPS处理器运行正确。



图14-22 小型SOPC运行后的7段数码管显示效果

14.6 测试二——UART实验

14.6.1 测试内容

本测试的主要内容是OpenMIPS处理器通过UART输出数据给PC，输出的数据从0x01依次递增至0xFF，然后再从0x00重新开始。

从本测试开始，不再给出详细的测试步骤，只是给出测试程序的说明、测试效果，而详细的测试步骤可以参考上一节GPIO实验。

14.6.2 测试程序

测试程序如下，源文件是本书附带光盘Code\Chapter14\uart_test目录下的inst_rom.S文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
.global _start

_start:
#####
##### 第一段 #####
#####

lui $1,0x1000
ori $1,$1,0x0003
ori $2,$0,0x80
sb  $2,0x0($1)      # 向地址0x10000003写入0x80
```

第一段

```
# 0x10000003对应UART控制器的Line Control Register
```

```
lui $1,0x1000  
ori $1,$1,0x0001  
ori $2,$0,0x00  
sb $2,0x0($1)      # 向地址0x10000001写入0x00, 此时对应的是分频系数的高字节
```

```
lui $1,0x1000  
ori $1,$1,0x0000  
ori $2,$0,0xB0  
sb $2,0x0($1)      # 向地址0x10000000写入0xB0, 此时对应的是分频系数的低字节
```

```
lui $1,0x1000  
ori $1,$1,0x0003  
ori $2,$0,0x03  
sb $2,0x0($1)      # 向地址0x10000003写入0x03  
# 0x10000003对应UART控制器的Line Control Register  
# 设置LCR为0x03, 表示8位数据位、没有奇偶校验位、1位停止位
```

```
#####
```

第二段

```
#####
```

```
ori $3,$0,0x0
#####
#loop1:
addi $3,$3,0x1
lui $1,0x1000
ori $1,$1,0x0000
sb $3,0x0($1) # 向地址0x10000000写入寄存器$3的最低字节
# 0x10000000对应UART控制器的Transmitting Holding Register
# 此处就是通过UART发送数据
```

第三段

```
#####
#loop2:
lui $1,0x1000
ori $1,$1,0x0005
lb $2,0x0($1) # 获取地址0x10000005处的值,
# 0x10000005对应UART控制器的Line Status寄存器
andi $2,$2,0x20
beq $2,$0,_loop2 # 判断LS寄存器的第5bit, 即“发送FIFO空标志”是否为1,
# 如果为1, 表示UART数据发送完毕, 可以接着发送下
```

第四段

```
一个数据了，  
    # 反之，回到loop2，等待发送完毕  
nop  
j _loop1           # 回到第三段，发送下一个数据  
nop
```

上述代码可以分为四段理解。

第一段：初始化UART控制器，包括设置分频系数、数据格式等，有三小步。

- 首先，向地址0x10000003写入0x80，UART控制器连接到Wishbone总线互联矩阵的从设备接口1，所以其地址是从0x10000000开始，参考表13-7可知，地址0x10000003对应的是UART控制器的Line Control Register，设置LCR的值为0x80，也就是设置LCR的最高位为1。设置完成后，地址0x10000000、0x10000001对应的就是两个分频系数寄存器。
- 接着，设置分频系数，此处设计UART的波特率为9600bps，系统时钟为27MHz，参考13.4.2节给出的分频系数计算公式，得到分频系数=27000000/(16*9600)，取整后为0xB0，所以设置分频系数的高字节为0x0，低字节为0xB0。
- 最后，设置LCR的值为0x03，参考表13-10对LCR寄存器的说明可知，此处就是设置UART收、发数据格式为8位数据位、没有奇偶校验位、1位停止位。

第二段：经过上面的步骤，UART控制器已经初始化完毕，可以收、发数据了。本段将寄存器\$3的值初始化为0x0。

第三段：这一段是通过UART发送数据的主循环，将寄存器\$3的值加1，然后将寄存器\$3的最低字节写入地址0x10000000，参考表13-7可知，地址0x10000000对应的是UART控制器的Transmitting Holding Register，写入其中的数据会被UART控制器发送出去。

第四段：检查UART控制器是否发送数据完毕，如果发送完毕，那么回到第三段，将寄存器\$3加1，再次通过UART发送，否则，等待数据发送完毕。其中，检查是否发送完毕的方法就是读取Line Status寄存器的值，参考表13-11可知，Line Status寄存器的第5bit是发送FIFO空标志，如果数据发送完毕，那么会设置该位为1。

在通过UART发送数据时注意，虽然UART控制器具有了FIFO，但是最好也不要连续快速发送数据，否则容易发生FIFO满的情况，导致数据丢失。所以，在测试程序中，在发送数据前都要先判断发送FIFO是否为空。

14.6.3 测试效果

将上一小节的测试程序编译后，写入Flash，然后下载14.3节得到的小型SOPC的配置文件到FPGA，详细步骤可以参考GPIO实验。

打开串口程序，将参数设置为与小型SOPC的一样，即设置波特率为9600bps、8位数据位、没有奇偶校验位、1位停止位。通过拨动开关SW17复位OpenMIPS，然后启动OpenMIPS，串口程序会得到如图14-23所示的结果，其中显示接收到的数据从0x01递增至0xFF，然后又从0x00开始，可知UART实验成功。

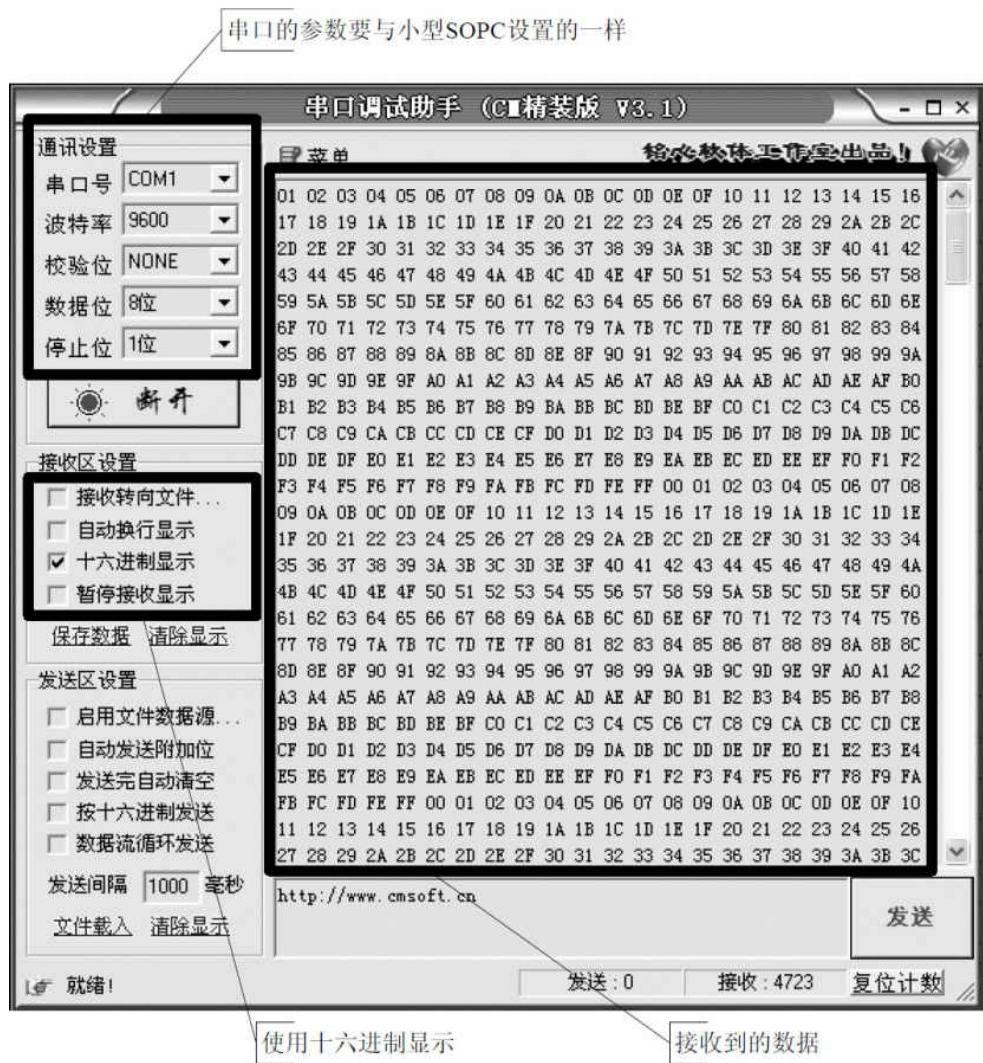


图14-23 UART实验的结果

14.7 测试三——模拟操作系统的加载过程

14.7.1 测试内容

本测试是一个稍微综合的测试，用来模拟操作系统的加载过程，其中需要编写两个程序：BootLoader、SimpleOS。它们在Flash中的存放位置如图14-24所示。

BootLoader存放在Flash从0x0处开始的空间，SimpleOS存放在Flash从0x304处开始的空间，另外，在Flash的0x300处存放的是SimpleOS的长度信息。

OpenMIPS启动后，会首先执行BootLoader。BootLoader读取存放在Flash的0x300处的长度信息length，根据该信息，将Flash从0x304处开始的length个字，依次复制到SDRAM从0x0处开始的空间，也就是将SimpleOS读取到SDRAM。读取结束后，跳转到SDRAM的0x0地址，将控制权交给SimpleOS，这个过程模拟了目前一些操作系统的加载过程。

其中的SimpleOS是一个很简单的程序，实现了UART的回显。当PC通过UART发送数据给小型SOPC时，会引发UART控制器的中断，SimpleOS中的中断处理程序会读取传递过来的数据，然后回送给PC，从而实现UART的回显。从描述中可以发现，该程序还可以验证实践版OpenMIPS处理器的中断功能是否实现正确。

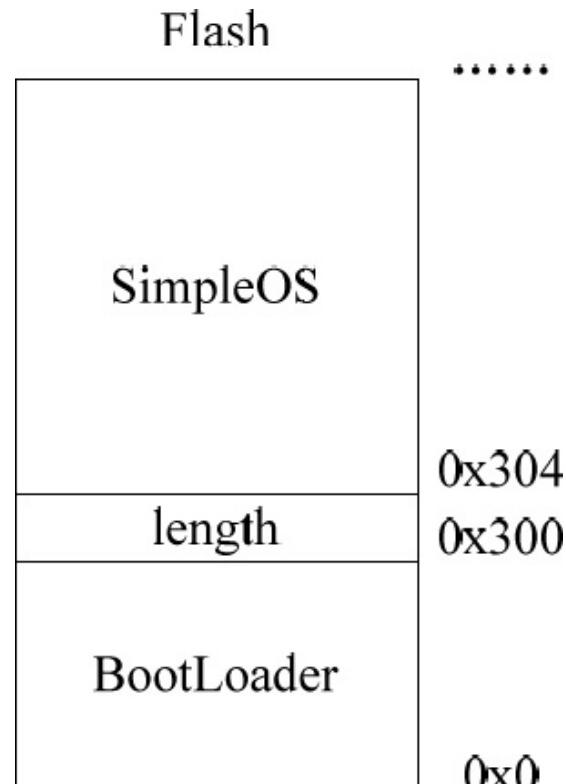


图14-24 BootLoader、SimpleOS两个程序在Flash中的存放位置

14.7.2 测试程序BootLoader

测试程序 BootLoader 的代码如下，读者可以在本书光盘 Code\Chapter14\bootloader 目录下找到源文件 BootLoader.S，以及编译所需的 Makefile、ram.ld 等文件。

```
.set noat
.set noreorder
.set nomacro

.org 0x0
.text
.align 4
.global _start

_start:
#####UART Controller Initialization#####
#####UART Controller Initialization#####

        lui $1, 0x1000
        ori $1, $1, 0x0003
        ori $2, $0, 0x80
        sb  $2, 0x0($1)

        lui $1, 0x1000
        ori $1, $1, 0x0001
        ori $2, $0, 0x00
```

```
sb $2,0x0($1)      # 设置分频系数的高字节

lui $1,0x1000
ori $1,$1,0x0000
ori $2,$0,0xB0
sb $2,0x0($1)      # 设置分频系数的低字节

lui $1,0x1000
ori $1,$1,0x0003
ori $2,$0,0x03
sb $2,0x0($1)      # 设置数据格式为8位数据位、没有奇偶校验位、1位停
止位
```

```
#####
第二段：GPIO 模块初始化
#####
```

```
lui $1,0x2000
ori $1,$1,0x0008
lui $2,0xffff
ori $2,$2,0xffff
sw $2,0x0($1)      # 使能所有GPIO输出端口

lui $1,0x2000
ori $1,$1,0x000C
lui $2,0x0000
ori $2,$2,0x0000
sw $2,0x0($1)      # 禁止GPIO输入中断
```

```
##### 第三段：等待SDRAM初始化完毕
```

```
#####
```

```
_waiting_sdram_init_done:
```

```
    lui $1,0x2000  
    ori $1,$1,0x0000  
    lw $4,0x0($1)      # 获取GPIO的输入  
    srl $4,$4,0x10  
    andi $4,$4,0x0001 # 判断第16位是否为1  
    beq $4,$0,_waiting_sdram_init_done  
    nop
```

```
##### 第四段：显示启动开始字符串
```

```
#####
```

```
    li $1,0x1  
    la $2,_BootBeginInfoStr      # 启动开始字符串的地址  
    la $3,_BootBeginInfoStrLen   # 启动开始字符串的长度  
    lb $5,0x0($3)  
  
1:  
    lb $4,0x0($2)                # 启动开始字符串中的字符  
    jal _print                   # 显示该字符  
    addi $2,$2,0x1               # 指向下一个字符  
    bne $5,$0,1b  
    subu $5,$5,$1
```

```
#####
```

第五段：获取 SimpleOS 的长度信息

```
#####
```

```
li $5, 0x4  
lui $1, 0x3000  
ori $1, $1, 0x0300  
lw $1, 0x0($1)      # 获取Flash的0x300处存放的SimpleOS  
长度信息  
# 保存到寄存器$1  
nop
```

```
#####
```

第六段：将 SimpleOS 复制到 SDRAM

```
#####
```

```
lui $2, 0x0000      # 寄存器$2指向SDRAM地址空间  
lui $3, 0x3000  
ori $3, $3, 0x0304      # 寄存器$3指向Flash中存放SimpleOS的  
地址  
1:  
lw $4, 0x0($3)      # 读取SimpleOS的内容  
nop  
sw $4, 0x0($2)      # 复制到SDRAM  
addi $2, $2, 0x4  
addi $3, $3, 0x4  
nop  
bgez $1, 1b          # 寄存器$1存储的是长度信息，每复制一个  
字，该值减4
```

```
# 当该值为0时，表示复制完毕  
subu $1,$1,$5  
  
##### 第七段：显示启动结束字符串 #####  
  
li $1,0x1  
la $2,_BootEndInfoStr      # 启动结束字符串的地址  
la $3,_BootEndInfoStrLen  # 启动结束字符串的长度  
lb $5,0x0($3)  
  
1:  
    lb $4,0x0($2)  
    jal _print  
    addi $2,$2,0x1  
    bne $5,$0,1b  
    sub $5,$5,$1  
  
    jr $0  
    nop  
  
##### 第八段：串口输出函数 #####  
  
_print:  
    lui $6,0x1000  
    ori $6,$6,0x0  
    sb $4,0x0($6)  
  
_waiting_transmit_done:  
    lui $6,0x1000
```

```
ori $6,$6,0x0005  
lb $7,0x0($6)      # 获取UART控制器中Line Status寄存器的值  
  
andi $7,$7,0x20      # 检查LS寄存器的第5bit“发送FIFO空标志”是否为1  
beq $7,$0,_waiting_transmit_done  
nop  
jr $31                # 返回，寄存器$31存储的是链接地址  
nop
```

第九段：一些预定义信息
#####

```
.data  
  
_BootBeginInfoStr:  
    .ascii "Loading OS into SDRAM...\\n"  
  
_BootBeginInfoStrLen:  
    .byte 26  
  
_BootEndInfoStr:  
    .ascii "Load OS into SDRAM DONE!!!\\n"  
  
_BootEndInfoStrLen:  
    .byte 28
```

上述程序可以分为九段理解，分别如下。

第一段：初始化UART控制器，包括设置分频系数、数据格式。
与UART实验中是一样的，读者可以参考14.6.2节。

第二段：初始化GPIO模块，包括使能所有输出接口、禁止输入中断。与GPIO实验中是一样的，读者可以参考14.5.2节。

第三段：SDRAM在使用之前需要初始化，此处就是等待SDRAM初始化完毕。在小型SOPC实现的时候将SDRAM控制器的输出信号sdram_init_done作为GPIO模块输入信号的第16bit，可以参考13.7节，所以此处读取GPIO的输入，然后判断第16bit是否为1，如果为1，表示SDRAM初始化完毕，否则，SDRAM没有初始化完毕。初始化完毕后，程序可以往后执行。

第四段：通过UART显示一些启动开始信息，其中`_BootBeginInfoStr`、`_BootBeginInfo StrLen`，都是在第九段中定义的，前者是要显示的启动开始信息，是一个字符串，后者是这个字符串的长度。UART发送数据是通过调用函数`printf`实现的，该函数在第八段中定义。注意在这里使用的指令`li`、`la`，这两条指令都是汇编指令，是汇编器定义的指令，分别等价于如下机器指令。

// li指令用来加载立即数到寄存器

li \$1,0x1 等价于 ori \$1,\$0,0x1

// la指令用来将指定的地址加载到寄存器，如下，其中%hi(addr)表示addr的高16bit

// %lo(addr)表示addr的低16bit

```
la    $2,_BootBeginInfoStr          等价于          lui    $2,  
%hi(_BootBeginInfoStr)  
  
                                addiu $2, $2,  
%lo(_BootBeginInfoStr)
```

第五段：读取Flash地址0x300处的字，也就是SimpleOS的长度信息。

第六段：将Flash从地址0x304开始的SimpleOS复制到SDRAM。

第七段：通过UART显示一些启动结束信息，其中_BootEndInfoStr、_BootEndInfoStrLen，都是在第九段中定义的，前者是要显示的启动结束信息，是一个字符串，后者是这个字符串的长度。

第八段：串口输出函数，要输出的字符就是寄存器\$4的最低字节，将其写入UART控制器的Transmitting Holding Register，然后等待发送完毕，最后返回。注意，因为调用该函数的时候使用的是jal指令，会将返回地址保存在寄存器\$31中，所以可以直接使用jr \$31指令返回。读者如果忘记了jal指令的用法，请复习一下8.2节。

第九段：一些预定义信息。

14.7.3 测试程序SimpleOS

测试程序SimpleOS的代码如下，读者可以在本书光盘中Code\Chapter14\simpleos目录下找到源文件SimpleOS.S，以及编译所需的Makefile、ram.ld等文件。

```
.org 0x0
.set noat
.set noreorder
.set nomacro
```

```
.global _start
```

```
_start:
```

```
#####
```

第一段：跳转到 0x100 处

```
#####
```

```
ori $1,$0,0x100
```

```
jr $1
```

```
#####
```

第二段：中断处理函数

```
#####
```

```
.org 0x20
```

```
mfc0 $1,$13,0x0      # 获取Cause寄存器的值
```

```
andi $4,$1,0x0800    # 判断是否是UART中断
```

```
bne $4,$0,_int2      # 如果是UART中断，那么转移到_int2进行处理
```

```
nop
```

```
eret      # 中断返回
```

```
#####
```

第三段：初始化 UART 控制器

```
#####
```

```
.org 0x100
```

```
lui $1,0x1000
```

```
ori $1,$1,0x0003
```

```
ori $2,$0,0x80
```

```
sb $2,0x0($1)
```

```
lui $1, 0x1000
ori $1,$1, 0x0001
ori $2,$0, 0x00
sb  $2, 0x0($1)      # 设置分频系数的高字节

lui $1, 0x1000
ori $1,$1, 0x0000
ori $2,$0, 0xB0
sb  $2, 0x0($1)      # 设置分频系数的低字节

lui $1, 0x1000
ori $1,$1, 0x0003
ori $2,$0, 0x03
sb  $2, 0x0($1)      # 设置数据格式为8位数据位、没有奇偶校验位、1位
停止位

lui $1, 0x1000
ori $1,$1, 0x0001
ori $2,$0, 0x01
sb  $2, 0x0($1)      # 向地址0x10000001写入0x01,
                      # 对应UART控制器的Interrupt Enable寄存器
                      # 将其设置为0x01，表示使能数据接收中断，参考表13-9
```

#####

第四段：使能 UART 中断

#####

```
lui $1,0x1000
ori $1,$1,0x0801
mtc0 $1,$12,0x0      # 设置CP0中的Status寄存器为0x1000801，使能
UART中断

_loop:
j _loop                # 循环等待UART中断发生
nop

#####
第五段：回显UART接收到的数据
#####

_int2:
lui $1,0x1000
ori $1,$1,0x0005
lb  $3,0x0($1)        # 通过Line Status寄存器检查是否有数据输入

andi $3,$3,0x01
beq $3,$0,_end        # 如果没有数据输入，就直接转移到_end
nop

_sendback:             # 如果有数据，那么先读取数据，然后再通过UART发送
回去
lui $1,0x1000
ori $1,$1,0x0000
lb  $2,0x0($1)        # 读取数据
```

```

sb $2, 0x0($1)      # 发送数据

loop2:                # 等待发送完毕
    lui $1, 0x1000
    ori $1,$1, 0x0005
    lb $2, 0x0($1)
    andi $2,$2, 0x20
    beq $2,$0, _loop2
    nop

    lui $1, 0x1000
    ori $1,$1, 0x0005
    lb $3, 0x0($1)      # 再次通过Line Status寄存器检查是否有数据输入
    andi $3,$3, 0x01
    bne $3,$0, _sendback # 如果有数据，那么再次回到_sendback处，继续回送数据
    nop

_end:
    eret                # 没有数据输入后，调用eret返回
    nop

```

上述程序可以分为五段理解，分别如下。

第一段：跳转到地址0x100处，因为SimpleOS是在SDRAM中运行的，而SDRAM挂接在Wishbone总线互联矩阵的从设备接口0，所以

SDRAM的起始地址是0x0。又因为OpenMIPS定义的异常处理例程入口地址为0x20、0x40（参考表11-2），所以尽量不要使用低地址空间，此处直接转移到0x100处开始执行。

第二段：在地址0x20处，定义了中断处理例程。其中获取CP0中Cause寄存器的值，依据Cause[11]的值判断是否是UART中断，在第13章设计SOPC的时候，将UART控制器的中断输出uart_int连接到了OpenMIPS处理器的中断输入接口，如下。

```
assign int = {3'b000, gpio_int, uart_int, timer_int};
```

上面中断输入接口int的值最终会被赋给Cause寄存器的IP[7:2]字段（参考10.3节协处理器CP0的实现），IP[3]就对应uart_int，参考表10-6可知，IP[3]是Cause寄存器的第11bit，所以此处依据Cause[11]的值判断是否是UART中断。如果是UART中断，那么转移到函数_int2进一步处理。

第三段：是地址0x100处的程序，在其中初始化UART控制器，包括设置分频系数、数据格式，读者应该很熟悉了，唯一增加的一点是设置Interrupt Enable Register（IER），该寄存器的地址是0x10000001，参考表13-9可知，设置其值为0x01，就是使能UART控制器的数据接收中断。

第四段：初始化CP0中的Status寄存器，设置其值为0x10000801，参考表10-5可知，即设置标志位IE为1、标志位IM[3]为1，对于小型SOPC而言，标志位IM[3]对应的就是UART中断，所以此处就是使能UART中断。第四段的最后会进入一个无限循环，等待UART中断的发生。

第五段：定义了函数_int2，该函数在第二段的中断处理例程中会被调用。首先获取UART控制器的Line Status寄存器的值，判断最低位是否为1，参考表13-11可知，该位为1表示接收FIFO不为空，有数据，该位为0表示接收FIFO为空，没有数据。如果没有数据，那么直接转移到_end处，中断处理结束。如果有数据，那么先读取数据，然后将该数据再通过UART发送出去。接下来等待数据发送完毕，其过程与UART实验是一样的。数据发送完毕后，再次判断是否接收到数据，如果没有数据，那么直接转移到_end处，中断处理结束。如果有数据，那么再次读取数据，通过UART发送出去。如此反复，直到没有接收数据为止。

因为SimpleOS是在SDRAM中运行，所以其链接脚本ram.ld与本章其余实验不同，其中设置起始地址为0x00000000，而不再是0x30000000，主要修改如下，完整文件可以参考本书附带光盘中Code\Chapter14\simpleos目录下的ram.ld。

```
MEMORY
{
    ram      : ORIGIN = 0x00000000, LENGTH =
0x00001000
}
```

14.7.4 将测试程序写入Flash

将上两小节介绍的程序分别编译，得到两个二进制文件：BootLoader.bin、SimpleOS.bin，然后需要将这两个文件按照图14-24所示的要求写入Flash，有两种方法。

(1) 分两次分别将BootLoader.bin、SimpleOS.bin写入Flash，写入完成后，还要将SimpleOS的长度信息写入Flash的0x300处。所以，一共需要写三次Flash。

(2) 将BootLoader.bin、SimpleOS.bin、SimpleOS的长度信息按照图14-24所示组合成一个二进制文件，然后写入Flash，这样只需写一次。

此处只介绍第(2)种方法，在本书附带光盘中Code\Chapter14目录下提供了一个小程序BinMerge.exe，该程序也是在Ubuntu下运行的，其作用就是将BootLoader.bin、SimpleOS.bin、SimpleOS的长度信息按照图14-24所示组合成一个二进制文件。

将BootLoader.bin、SimpleOS.bin复制到BinMerge.exe同一个目录下，然后输入如下命令。其中-f后面的参数是简单操作系统的二进制文件，此处就是SimpleOS.bin，-o后面的参数是最终输出的二进制文件名，此处命名为Image.bin，读者可以依据实际情况命名。

```
./BinMerge.exe -f SimpleOS.bin -o Image.bin
```

还可以将上述命令放在SimpleOS对应的Makefile文件中，这样在编译SimpleOS的时候，就直接得到Image.bin。为此，修改SimpleOS对应的Makefile文件如下。完整文件可以参考本书附带光盘中Code\Chapter14\simpleos目录下的Makefile。

```
ifndef CROSS_COMPILE  
CROSS_COMPILE = mips-sde-elf-  
endif  
CC = $(CROSS_COMPILE)as
```

```
LD = $(CROSS_COMPILE)ld
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump

OBJECTS = SimpleOS.o

export CROSS_COMPILE

# ****
# Rules of Compilation
# ****

# 在目标中添加Image.bin
all: SimpleOS.om SimpleOS.bin SimpleOS.asm SimpleOS.data
    SimpleOS.mif Image.bin

%.o: %.S
    $(CC) -mips32 $< -o $@

SimpleOS.om: ram.ld $(OBJECTS)
    $(LD) -T ram.ld $(OBJECTS) -o $@

SimpleOS.bin: SimpleOS.om
    $(OBJCOPY) -O binary $< $@

SimpleOS.asm: SimpleOS.om
    $(OBJDUMP) -D $< > $@
```

```
# 生成Image.bin, 实际就是调用BinMerge.exe程序,  
  
# 注意要将BootLoader.bin复制到SimpleOS.s所在目录  
  
Image.bin: SimpleOS.bin  
  
. ./BinMerge.exe -f $< -o $@  
  
clean:  
    rm -f *.o *.om SimpleOS.bin *.data *.mif *.asm
```

得到Image.bin后，需要将其写入Flash，本节不再给出详细的写入步骤，读者可以参考GPIO实验。

14.7.5 测试效果

将Image.bin写入Flash，然后下载14.3节得到的小型SOPC的配置文件到FPGA，详细步骤参考GPIO实验。

打开串口程序，将参数设置为与小型SOPC的一样，即设置波特率为9600bps、8位数据位、没有奇偶校验位、1位停止位。通过拨动开关SW17复位OpenMIPS，然后启动OpenMIPS，串口程序会得到如图14-25所示的结果，可知，BootLoader加载操作系统成功。然后，PC通过串口发送任意字符，都会被小型SOPC回送，如图14-26所示，可知SimpleOS工作正常。

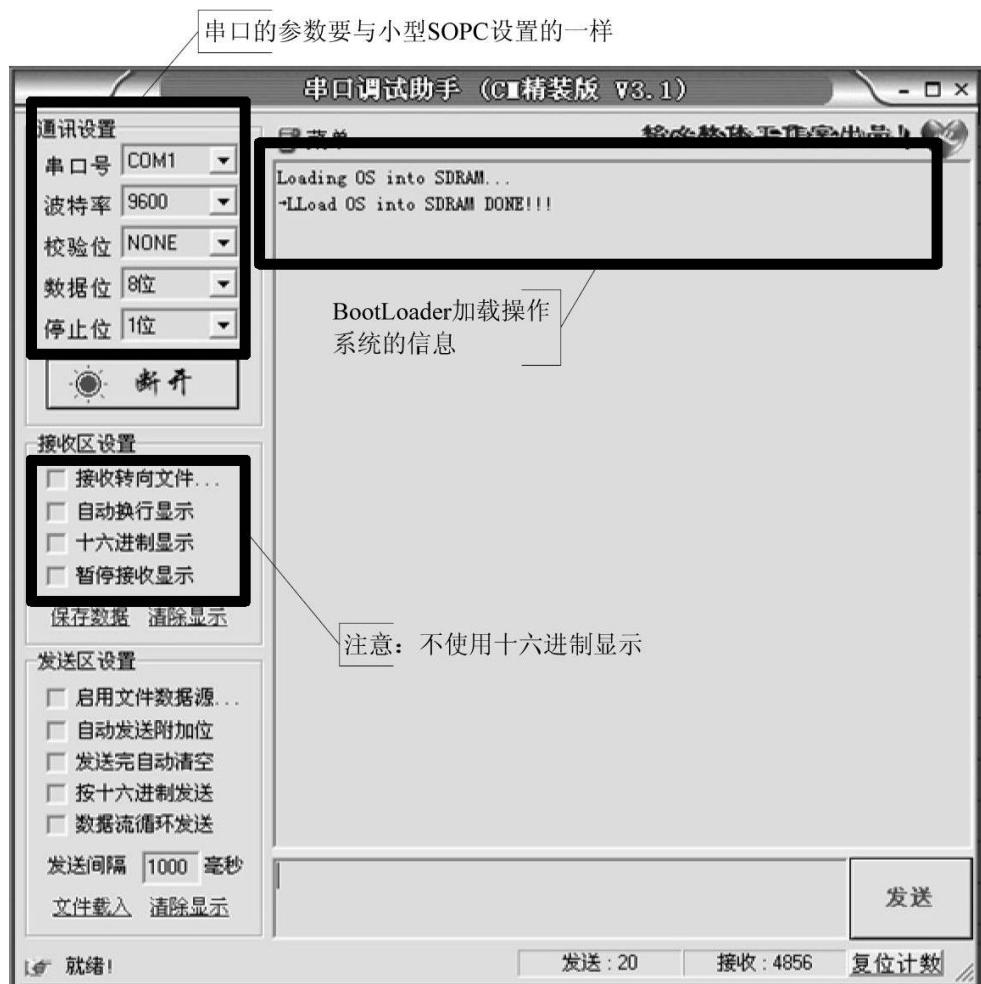


图14-25 BootLoader加载操作系统成功

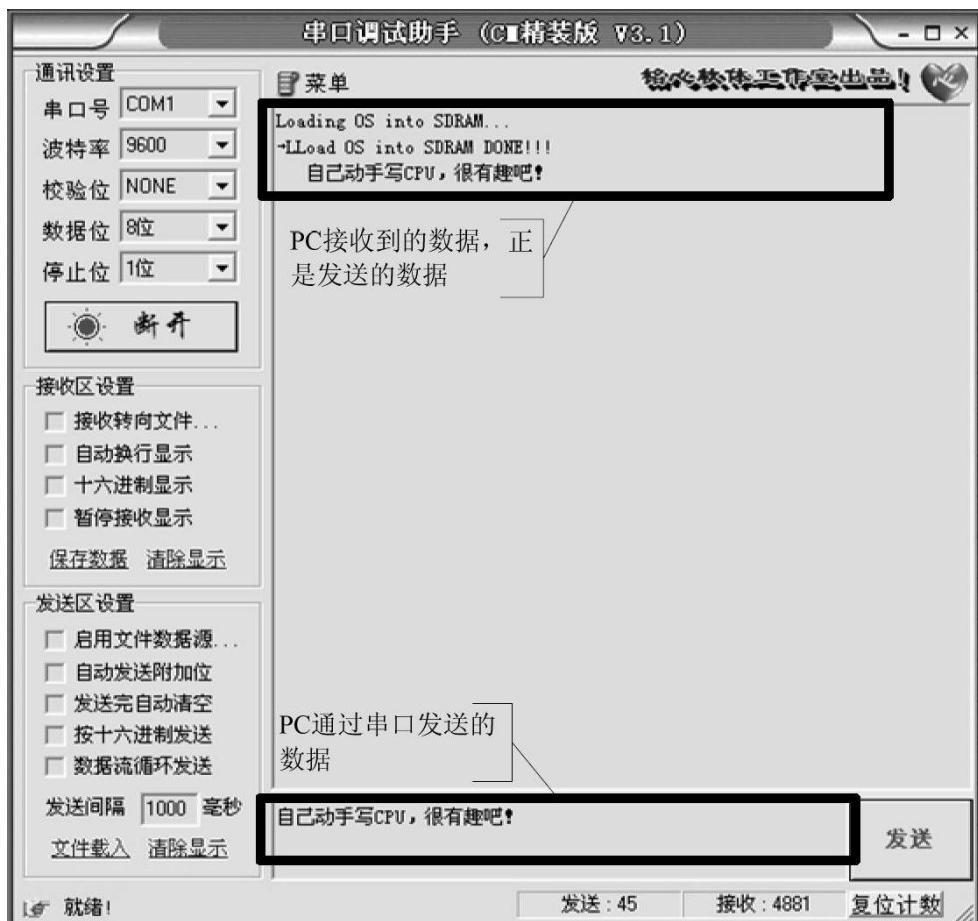


图14-26 SimpleOS工作正常

14.8 本章小结

本章将第13章实现的小型SOPC下载到开发平台DE2，并通过三个实验——GPIO实验、UART实验、模拟操作系统加载过程实验，证明了实践版OpenMIPS处理器设计正确，并且小型SOPC的各个模块也都运行正确。

第15章将为实践版OpenMIPS处理器移植嵌入式操作系统μC/OS-II。

第15章 为OpenMIPS处理器移植 μC/OS-II

截止到本章，我们已经实现了实践版OpenMIPS处理器，并且以OpenMIPS为核心，实现了一个小型SOPC，包括：GPIO、UART控制器、Flash控制器、SDRAM控制器。应该可以说，一个属于用户自己的小型计算机诞生了。在第14章我们还编写程序运行在这个小型计算机上，实现了输入/输出、串口发送数据等功能，但似乎还缺少什么，是的，缺少一个操作系统，作为一个完整的小型计算机，怎么能够没有属于自己的操作系统呢？现在就为OpenMIPS处理器打造属于自己的操作系统。

本章的目的是移植开源实时操作系统μC/OS-II到OpenMIPS，首先讨论了为什么需要操作系统、什么是实时操作系统，然后介绍了开源实时操作系统μC/OS-II，主要从其特点、重要概念、基本功能、文件体系、移植条件等几个方面讲解。接着详细给出了移植μC/OS-II到OpenMIPS的步骤，最后，通过DE2开发平台验证移植后的μC/OS-II工作正确，从而证明移植成功。

15.1 为什么需要操作系统

邹恒明在《计算机的心智：操作系统之哲学原理》一书中有如下观点。

人有心智吗？我想所有人都会回答：有！人的心智就是人的灵气。这是每一个人的生命之气。就是这个灵气赋予了人丰富的思维、感受和行动能力。

那么计算机有心智吗？这不是一个诡秘或者搞笑的问题。

人们通常认为能够运动的生命都是有灵气的，既然计算机能够完成一些人脑才能够完成的理性任务，它当然也有心智！而这个心智就是操作系统。因为操作系统赋予了计算机以活力。

.....

操作系统作为计算机赖以运转的控制中心，称其为计算机的心智可谓恰如其分。

处理器、存储器、I/O等硬件设备组成计算机的躯壳，操作系统构成计算机的心智，没有心智的计算机犹如人之没有魂灵，这个比喻有点诗意了，但是也印证了操作系统的重要性。

操作系统是介于计算机和应用软件之间的一个软件系统，用于掌控计算机上的所有事情，其下是硬件平台，其上是应用软件。如图15-1所示。



图15-1 操作系统上下界面

还是借用邹恒明在《计算机的心智：操作系统之哲学原理》一书中的观点，其认为操作系统扮演两个根本角色：魔幻家和管理者。

1. 魔幻家角色

操作系统将计算机以一种更加容易、更加方便、更加强大的方式呈现给用户使用。直白的说，就是把差的东西变好，把少的东西变多，把复杂的东西变得容易。例如，如果在裸机上直接编程是很困难的，因为各种数据转移均需要用户自己来控制，对不同设备要用不同的命令来驱动，而这对一般人来说很难胜任。操作系统将这些工作从用户手中接过来，从而让用户感觉到编程是一件容易的事。

2. 管理者角色

操作系统管理计算机上的软硬件资源，具体包括CPU、内存、外存、I/O等。操作系统使得不同用户之间或者同一用户运行的不同程序之间可以安全有序的共享这些硬件资源。管理的关键原则是有效和公平，有效指的是不能浪费资源，公平指的是每个人都有享有资源的可能。

正是因为操作系统扮演的这两个角色，可以让应用程序脱离硬件，提高应用程序的可移植性和可读性，另外，当要实现的应用比较复杂时，操作系统可以为这个复杂的应用提供管理机制，程序员只要完成功能函数，并且添加任务即可，不用再去处理不同任务之间的通信以及各个不同功能之间如何协同工作等问题，所以我们需要操作系统。

15.2 嵌入式实时操作系统介绍

既然操作系统如此重要，那我们就给OpenMIPS安装Windows好了。

.....

当然不行。首先，操作系统要能够在新的处理器上运行，是需要修改部分代码的，而Windows没有公开源代码，无法修改。其次，Windows太庞大了，而我们DE2上的Flash只有4MB、SDRAM只有8MB，所以不能运行Windows。实际上，一般在SOPC上运行的都是一些小巧的嵌入式实时操作系统（Embedded Real-time Operation System）。从名称上可以发现是嵌入式操作系统、实时操作系统的交集。所以下面分别介绍这两种操作系统。

1. 嵌入式操作系统

在解释什么是嵌入式操作系统之前，需要先解释什么是嵌入式系统。目前，我国普遍认同的嵌入式系统定义为：以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗等严格要求的专用计算机系统。

与个人计算机这样的通用计算机系统不同，嵌入式系统通常执行的是带有特定要求的、预先定义的任务。由于嵌入式系统只针对一些特定任务，所以设计人员能够对它进行优化，减小尺寸，降低成本。

嵌入式操作系统（EOS： Embedded Operating System）就是用于嵌入式系统的操作系统。

2. 实时操作系统

实时操作系统（RTOS：Real Time Operating System）是指当外界事件或数据产生时，能够接收并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程、做出快速响应，并控制所有实时任务协调一致运行的操作系统。因而，提供及时响应和高可靠性是其主要特点。实时操作系统有硬实时和软实时之分，硬实时要求在规定的时间内必须完成操作，这是在操作系统设计时保证的；软实时则只要按照任务的优先级，尽可能快地完成操作即可。

RTOS的内核一般采用可剥夺型内核（Preemptive Kernel），在这种内核中，当有更高优先级的任务就绪时，总能得到CPU的控制权。所以，在可剥夺型内核中，最高优先级的任务何时可以执行、何时可以得到CPU的控制权，是可知的。

目前，常用的嵌入式实时操作系统有μC/OS-II、RTEMS、VxWorks、eCos、FreeRTOS、RTLinux、T-Kernel等，国内目前也有很多，比如Raw OS、RT-Thread等。本章计划为OpenMIPS处理器移植μC/OS-II，所以下面仅对μC/OS-II进行详细介绍。

15.3 μC/OS-II简介

μC/OS-II读作“micro C O S 2”，意为“微控制器操作系统版本2”，是一个完整的、可移植、可裁剪、开源的、可剥夺型实时操作系统，μC/OS-II在世界范围内得到广泛使用，包括诸多领域，如手机、路由器、集线器、不间断电源、飞行器、医疗设备及工业控制等，并且得

到了美国航空管理局的认证，这充分证明了μC/OS-II的高度稳定可靠，能够用于安全性条件极为苛刻的系统。

μC/OS-II采用ANSI的C语言编写，包含一小部分汇编语言代码，使之可以在不同架构的微处理器上使用。至今，从8位到32位，μC/OS-II已经在超过40种不同架构的微处理器上运行。

μC/OS-II 的 作 者 Jean J. Labrosse 出 版 了 介 绍 μC/OS-II 的 书 《MicroC/OS-II The Real Time Kernel》，该书至今已出了第2版，是学 习 μC/OS-II 的 必 备 书 籍，其 中 文 版 由 邵 贝 贝 翻 译，书 名 为 《嵌 入 式 实 时 操 作 系 统 μC/OS-II（第2版）》，读 者 朋 友 可 以 参 考。

15.4 μC/OS-II特点

μC/OS-II具有以下特点。

1. 提供源代码

与Linux一样，μC/OS-II的源代码也是开放的，用户可以在<http://micrium.com/>网站下载所有源代码。另外，《嵌入式实时操作系统μC/OS-II（第2版）》一书的附带光盘中也有μC/OS-II V2.52版本的所有源代码。该源代码清晰易懂，且结构合理，Jean J.Labrosse也提供了详尽的注解。

2. 可移植 (PortabIe)

μC/OS-II源代码的绝大部分是使用移植性很强的ANSI C编写的，只有与微处理器硬件相关的部分是使用汇编语言编写的。汇编语言编

写的部分已经压缩到最低限度，以使μC/OS-II便于移植到其他微处理器上。

3. 可固化 (ROMabIE)

μC/OS-II是为嵌入式应用而设计的操作系统，只要具备合适的软硬件工具，就可将μC/OS-II嵌入到产品中，成为产品的一部分。

4. 可裁剪 (ScalabIE)

可根据应用的需要来裁剪系统功能。可以只使用μC/OS-II中应用程序需要的系统服务，也可以使用μC/OS-II的所有功能，从而灵活控制μC/OS-II占用的存储器空间。包含全部功能的核心部分代码占用8.3KB，经过裁剪，最少仅为2.7KB。

5. 可剥夺型 (Preemptive)

μC/OS-II是完全可剥夺型的实时内核，总是运行优先级最高的就绪任务。

6. 多任务

μC/OS-II V2.52版本可管理64个任务，一般情况下，建议保留8个任务给μC/OS-II，这样，留给用户应用程序的任务最多可有56个。系统赋予每个任务的优先级必须是不同的，这意味着μC/OS-II不支持时间片轮转调度法 (Round-robin Scheduling)。

最新的μC/OS-II V2.91版本可以管理多达250个任务。

7. 可确定性

μ C/OS-II中的绝大多数函数调用和服务的执行时间具有确定性，也就是说，用户总是能知道 μ C/OS-II的函数调用与服务执行了多长时间。

8. 任务栈

每个任务都有自己单独的栈， μ C/OS-II允许每个任务有不同的栈空间，以便满足应用程序对RAM的需求，使用 μ C/OS-II的栈空间校验函数可确定每个任务到底需要多少栈空间。

9. 系统服务

μ C/OS-II提供很多系统服务，例如：信号量、事件标志、消息邮箱、消息队列、块大小固定的内存申请与释放、时间管理函数等。

10. 中断管理

中断可使正在执行的任务暂时挂起，如果优先级更高的任务被中断唤醒，那么高优先级的任务会在中断嵌套全部退出后立即执行，中断嵌套层数最多可达255层。

11. 稳定性与可靠性

μ C/OS-II是基于 μ C/OS的， μ C/OS自1992年发布后，已有数百个商业应用。 μ C/OS-II与 μ C/OS的内核是一样的，只是提供了更多的功能。另外，2000年7月， μ C/OS-II在一个航空项目中得到了美国联邦航空管理局（FAA：Federal Aviation Administration）对用于商用飞机的、符合RTCA DO-178B标准的认证，该标准对用于航空设备方面的软件提出了要求。为了符合这一标准，必须尽可能地通过文件描述和测试，展示软件在稳定性与安全性这两方面都符合要求。这一结论对于操作

系统来说特别重要，因为这一结论表明，该操作系统的质量得到了认证，可以在任何应用中使用。 μ C/OS-II的每一种功能、每一个函数及每一行代码都经过了考验与测试。在2011年发射到火星的“好奇号”火星探测车上就有一个分析实验室由 μ C/OS-II控制。

15.5 μ C/OS-II的几个概念

15.5.1 任务

任务，也称为线程，是一个简单的程序。 μ C/OS-II是一个多任务的操作系统。典型的是，每个任务都是一个无限循环，都可能处在以下五种状态之一——休眠态、就绪态、运行态、挂起态、中断态。

- 休眠态（Dormant）：相当于任务驻留在内存中，但并不被内核所调度。
- 就绪态（Ready）：意味着任务已经准备好，可以运行，但由于该任务的优先级比正在运行的任务的优先级低，所以暂时不能运行。任务一旦创建，就处于就绪态，准备运行。
- 运行态（Running）：是指任务掌握了CPU的使用权，正在运行中。处于就绪态的最高优先级的任务能够获得CPU的使用权，从而处于运行态。
- 挂起态（Pending）：也可称为等待事件态，是指任务在等待某一事件的发生（例如：等待某外设的I/O操作，等待某共享资源由不能使用变成能使用，等待定时脉冲的到来，等等）。此时，任务将被放在该事件的等待列表中。

- 中断态（Interrupt）：正在运行的任务可以被中断，除非该任务将中断关闭。任务被中断后，CPU将进入中断服务例程，被中断的任务进入中断态。

μ C/OS-II V2.91版本最多可管理250个任务，这些任务通常都是一个无限循环的函数。系统初始化时会自动创建两个任务：一个是空闲任务，其优先级最低，只是不停地给一个32位的整型变量加1；另一个是统计任务，该任务每秒运行一次，负责采集当前CPU的利用率。

每个任务对应一个任务控制块OS_TCB，任务控制块就是一个数据结构，当任务的CPU使用权被剥夺时，需要使用它来保存该任务的状态。其定义如下，读者不需要明白所有的内容，只需要知道任务控制块的第一个字中存储的是该任务的堆栈栈顶指针，在15.11.3节修改os_cpu_a.S文件时，需要有这个知识。

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr; /* 任务的堆栈栈顶指针 */
};

#if OS_TASK_CREATE_EXT_EN > 0u
    void           *OSTCBExtPtr;
    OS_STK          *OSTCBStkBottom;
    INT32U         OSTCBStkSize;
    INT16U         OSTCBOpt;

```

```
    INT16U          OSTCBId;

#endif

    struct os_tcb    *OSTCBNext;
    struct os_tcb    *OSTCBPrev;

#if (OS_EVENT_EN)
    OS_EVENT        *OSTCBEVENTPtr;
#endif

#if (OS_EVENT_EN) && (OS_EVENT_MULTI_EN > 0u)
    OS_EVENT        **OSTCBEVENTMultiPtr;
#endif

#if ((OS_Q_EN > 0u) && (OS_MAX_QS > 0u)) || (OS_MBOX_EN > 0u)
    void            *OSTCBMsg;
#endif

#if (OS_FLAG_EN > 0u) && (OS_MAX_FLAGS > 0u)
#if OS_TASK_DEL_EN > 0u
    OS_FLAG_NODE   *OSTCBFlagNode;
#endif
    OS_FLAGS        OSTCBFlagsRdy;
#endif

    INT32U          OSTCBDly;
```

```
INT8U          OSTCBStat;
INT8U          OSTCBStatPending;
INT8U          OSTCBPriority;

INT8U          OSTCBX;
INT8U          OSTCBY;
OS_PRIO        OSTCBBitX;
OS_PRIO        OSTCBBitY;

#if OS_TASK_DEL_EN > 0u
    INT8U          OSTCBDeleteReq;
#endif

#if OS_TASK_PROFILE_EN > 0u
    INT32U         OSTCBCTxSwCtr;
    INT32U         OSTCBCyclesTotal;
    INT32U         OSTCBCyclesStart;
    OS_STK         *OSTCBStackBase;
    INT32U         OSTCBStackUsed;
#endif

#if OS_TASK_NAME_EN > 0u
    INT8U          *OSTCBTaskName;
#endif

#if OS_TASK_REG_TBL_SIZE > 0u
    INT32U         OSTCBRegTbl[OS_TASK_REG_TBL_SIZE];

```

```
#endif  
} OS_TCB;
```

15.5.2 任务调度

调度（Dispatch）是内核的主要职责之一，就是决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的。每个任务根据其重要程度的不同，被赋予一定的优先级。基于优先级调度法是指，CPU总是让处于就绪态的、优先级最高的任务运行。

μ C/OS-II是可剥夺型实时多任务内核。可剥夺型实时内核在任何时候都运行就绪了的最高优先级的任务。 μ C/OS-II的任务调度是完全基于任务优先级的抢占式调度，也就是最高优先级的任务一旦处于就绪状态，就立即抢占正在运行的低优先级任务的CPU资源。为了简化系统设计， μ C/OS-II规定所有任务的优先级不同，因而任务的优先级也同时唯一标识了该任务本身。

15.5.3 任务切换

任务切换（Context Switch），有时也被称为上下文切换。当多任务内核决定运行另一个任务时，它首先保存正在运行的任务的状态（Context），即CPU全部寄存器的内容。这些内容保存在任务的堆栈中，然后把下一个将要运行的任务的状态从该任务的堆栈中重新装入CPU的寄存器，开始下一个任务的运行，这一过程叫做任务切换。

15.5.4 μC/OS-II的中断处理

中断发生后，一般会进入中断服务子程序，μC/OS-II的中断服务子程序要用汇编语言来编写，其结构如下。

```
保存CPU的全部寄存器  
调用函数OSIntEnter或者直接将变量OSIntNesting加1  
清除中断源  
重新开中断  
执行用户代码做中断处理  
调用函数OSIntExit  
恢复CPU的全部寄存器  
执行中断返回指令
```

中断可使正在执行的任务暂时挂起，所以，进入中断服务子程序后，首先应将CPU的全部寄存器保存到被中断的任务的堆栈中。μC/OS-II需要知道正在做中断服务，所以要调用函数OSIntEnter或者直接将全局变量OSIntNesting加1。这样，只要变量OSIntNesting不为0，就表示处于中断处理过程中。然后，可以选择是否允许新的中断，如果允许，那么必须清除中断源，重新开中断。

现在可以正式开始处理中断了。中断处理结束后，需要调用函数OSIntExit，用于将全局变量OSIntNesting减1，当OSIntNesting等于0时，就表示所有中断，包括嵌套的中断都已经处理完毕。此时，μC/OS-II必须判断是否有优先级更高的任务被中断服务子程序唤醒，如果有，就返回到更高优先级的任务，反之，返回到被中断的任务。将要运行的任务的寄存器从堆栈恢复。最后，执行中断返回指令。

15.5.5 时钟节拍

时钟节拍（Clock Tick）是特定的周期性中断，这个中断可以认为是系统心脏的脉动。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍源可以是专门的硬件定时器，也可以是来自50/60Hz交流电源的信号。 μ C/OS-II的节拍率应为每秒10-100次，或者说10-100Hz。时钟节拍率越高，系统的额外负荷就越重。时钟节拍的实际频率取决于用户应用程序的精度要求。

μ C/OS-II的启动过程中，一般先调用系统初始化函数OSInit，再调用系统启动函数OSStart。在调用OSStart之后做的第一件事就是允许时钟节拍中断，容易犯的错误是，将允许时钟节拍中断放在函数OSInit之后、函数OSStart之前。

```
void main(void)
{
    .....
    OSInit();           /* 初始化μC/OS-II */
    .....
    /* 应用程序初始化代码 */
    /* 通过调用函数OSTaskCreate创建至少一个任务 */
    .....
```

```
允许时钟节拍中断 /*  
错误！！不可以在此处允许时钟节拍中断  
*/  
.....  
OSStart(); /* 系统启动 */  
}
```

15.5.6 μC/OS-II的初始化

μC/OS-II在调用其他任何服务之前，首先要对系统初始化，这是通过调用系统初始化函数OSInit实现的，该函数会初始化μC/OS-II的所有变量和数据结构，并建立空闲任务OS_TaskIdle，该任务总是处于就绪态，优先级也总是设成最低，即OS_LOWEST_PRIO。如果允许建立统计任务，那么函数OSInit还会建立统计任务OS_TaskStat，也进入就绪态，优先级是次低的，即OS_LOWEST_PRIO-1。

15.5.7 μC/OS-II的启动

μ C/OS-II启动前，至少需要创建一个用户任务。 μ C/OS-II的启动是通过调用函数OSStart实现的。OSStart从任务就绪表中找出用户建立的优先级最高的任务的任务控制块TCB。然后调用高优先级就绪任务启动函数OSStartHighRdy，该函数位于文件os_cpu_a.S中。实质上，函数OSStartHighRdy的作用是将任务堆栈中保存的值恢复到对应的CPU寄存器，然后执行一条中断返回指令，就开始运行用户建立的优先级最高的任务，在15.11.3节会列出该函数的代码。

15.6 μ C/OS-II的基本功能

μ C/OS-II实际上是一个实时操作系统内核，只包含了任务调度、任务间的通信与同步、任务管理、时间管理、内存管理等基本功能，没有提供输入/输出管理、文件系统及网络等额外功能。其中任务调度在15.5.2节已介绍，所以下面简单介绍 μ C/OS-II的其余四项基本功能。

15.6.1 任务间的通信与同步

对于一个多任务操作系统而言，任务间的通信与同步是必不可少的。任务间的同步是指，异步环境下的一组并发执行任务因各自的执行结果互为对方的执行条件，因而，任务之间需互发信号，以使各任务按一定的速度执行。 μ C/OS-II中，任务或中断服务子程序可通过事件控制块ECB（Event Control Block）向另外的任务发信号，此处的信号（Signal）也就是事件（Event），可以是信号量、邮箱、消息队列等。

15.6.2 任务管理

任务管理包括建立任务、删除任务、改变任务的优先级、挂起和恢复任务等功能，通过一系列函数实现。任务管理的主要函数如下。

1. OSTaskCreate或OSTaskCreateExt，用于建立一个任务。
2. OSTaskStkChk，用于检验堆栈。堆栈是由连续的内存空间组成的，每个任务都有自己的堆栈。本函数用来检验堆栈空间的大小。
3. OSTaskDel，用于删除一个任务。其功能是将任务返回并使之处于休眠状态，这样系统将不再调度该任务。
4. OSTaskDelReq，用于请求删除一个任务。
5. OSTaskChangePrio，用于改变任务的优先级。任务建立时，系统为任务分配了一个优先级，之后，可通过调用本函数来动态改变任务的优先级。
6. OSTaskSuspend，用于挂起任务。
7. OSTaskResume，用于恢复被挂起的任务。
8. OSTaskQuery，用于获得自身或其他任务的信息。

15.6.3 时间管理

μ C/OS-II利用时钟节拍产生的周期性中断，实现延时和超时控制等功能。时间管理是通过一系列与时间有关的函数实现的。

1. OSTimeDly，任务延时函数，该函数会使 μ C/OS-II进行一次任务调度，并且执行下一个优先级最高的就绪态任务。任务调用该函数后，一旦规定的时间期满或者有其他任务通过调用函数OSTimeDlyResume取消了延时，它就会立即进入就绪态。
2. OSTimeDlyHMSM，按时、分、秒、毫秒延时的函数。与函数OSTimeDly不同之处在于，后者的延时单位是时钟节拍，而前者是按时、分、秒、毫秒来定义延时时间。其余功能是一样的。
3. OSTimeDlyResume，恢复延时任务的函数。通过调用本函数，可使指定任务不必等待延时期满，就可以处于就绪态。
4. OSTimeGet，时钟节拍发生时，会将一个计数器的值加1，本函数用来获得当前计数器的值。
5. OSTimeSet，本函数用来设置计数器的值。

15.6.4 内存管理

ANSI C中，一般使用malloc和free两个函数动态地分配和释放内存。这样，随着内存空间的不断分配和释放，就会把原来很大的一块连续内存区域逐渐地分割成许多非常小的但彼此之间又不相邻的内存块，也就是产生内存碎片问题。由于系统中大量内存碎片的存在，使得再有程序要求为之分配内存时，可能出现总的内存空间容量比所要求的大，但彼此不连续，也就是都以碎片的形式存在，导致内存分配

失败的情况。而且，由于内存管理算法上的原因，`malloc`和`free`函数的执行时间是不确定的，这在嵌入式实时操作系统中是非常危险的。

为了解决多次动态分配与释放内存所引起的内存碎片以及分配、释放函数执行时间不确定的问题，μC/OS-II把连续的大块内存按分区来管理。每个分区都包含整数个大小相同的内存块，但不同分区之间内存块的大小可以不同。需要动态分配内存时，可选择一个适当的分区，按块来分配内存；释放内存时，将该块放回它以前所属的分区。这样，就能有效解决内存碎片问题。而且，每次调用函数`malloc`和`free`进行分配和释放的都是整数倍的固定内存块长，这样执行时间就是确定的了。

μC/OS-II中使用内存控制块（Memory Control Blocks）的数据结构跟踪每一个内存分区，每个分区都有属于自己的内存控制块。内存管理的主要函数如下。

1. `OSMemCreate`，用于建立一个内存分区。
2. `OSMemGet`，用于分配一个内存块。当某一任务被调度执行时，必须先从已建立的内存分区中为该任务申请一个内存块。
3. `OSMemPut`，释放一个内存块。当某一任务不再使用一个内存块时，必须及时地把它放回到相应的内存分区中，以便下一次的分配操作。
4. `OSMemQuery`，用于查询一个特定内存分区的状态，如：查询内存分区中内存块的大小、可用内存块数、正在使用的内存块数等信息。

15.7 μC/OS-II的文件体系

μC/OS-II的文件体系如图15-2所示。读者需要注意的是，此处是以V2.91版本为例给出的文件体系，而《嵌入式实时操作系统μC/OS-II（第2版）》一书给出的文件体系是以V2.52版本为例，两者稍微有点不同，读者朋友在阅读的时候可以对比区分。

从图中可以发现，μC/OS-II的代码可以分为四部分。

- 与处理器无关的代码：这就是μC/OS-II内核的全部代码。
- 与具体处理器相关的代码：对每一种处理器，此处的代码都有区别，移植μC/OS-II主要就是修改此部分的代码。图15-2中，该部分的文件是参照移植到M14K处理器列出的，移植到其他处理器时，该部分的文件可能会有差别，比如本章将μC/OS-II移植到OpenMIPS处理器时，就没有其中的CPU_A.S文件。



图15-2 μC/OS-II的文件体系（以V2.91为例）

- 与应用相关的代码：与用户应用程序相关的头文件。
- 用户代码：用户编写的应用程序。

15.8 μC/OS-II的移植条件

移植就是使一个操作系统能够在某个微处理器平台或微控制器上运行。为了方便移植，μC/OS-II的大部分代码使用C语言编写，但是仍需要用C语言和汇编语言编写一些与处理器硬件相关的代码，这是因

为μC/OS-II在访问处理器的寄存器时，只能通过汇编语言来实现，这部分对应的也就是图15-2中的“与具体处理器相关代码”。由于μC/OS-II在设计之初就已经充分考虑了可移植性，所以μC/OS-II的移植相对来说是比较容易的。为了移植μC/OS-II，目标处理器必须满足以下条件。

1. 处理器的C编译器能产生可重入代码

可重入代码指的是可被多个任务同时调用，而不会破坏数据的一段代码。μC/OS-II是多任务内核，函数可能会被多个任务调用，代码的可重入性是保证完成多任务的基础。

可重入代码中不应该有全局变量或静态变量，因为这些变量会保存某一个进程的修改。可重入代码中的变量应该都是局部变量，每次重新调用时变量被重新赋值，从而保证，每个进程对它的调用都产生同样的结果。图15-3列举了两个函数作为例子，它们的区别在于变量temp不同，左边函数中的temp作为全局变量存在，右边函数中的temp作为局部变量存在，因此左边的函数是不可重入的，而右边的函数是可重入的。

```
int temp;

void swap(int * x, int * y)
{
    temp = *x;
    *x = *y;
    *y = temp;
}
```

```
void swap(int * x, int * y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

图15-3 不可重入代码与可重入代码示例

为了产生可重入代码，除了在C程序中使用局部变量，还需要C编译器的支持，本书一直在使用的MIPS编译器能生成可重入代码。

2. 用C语言可打开和关闭中断

OpenMIPS处理器的CP0中有Status寄存器，其最低位为中断使能标志IE，通过设置该位的值，能够打开、关闭中断。参考10.2节。

3. 处理器支持中断并且能产生定时中断

μ C/OS-II是通过处理器产生的时钟节拍中断来实现多任务调度的。而OpenMIPS处理器支持中断，并且能产生时钟中断，满足此处的要求。

4. 处理器支持能够容纳一定量数据的硬件堆栈

OpenMIPS的寻址空间达到4GB，完全能满足堆栈需求。

5. 处理器有将堆栈指针和CPU其余寄存器读出和存储到堆栈（或内存）的指令

μ C/OS-II进行任务调度时，会把当前任务的CPU寄存器存放到此任务的堆栈中，然后再从另一个任务的堆栈中恢复原来的工作寄存器，从而继续运行另一个任务，所以需要压栈、出栈指令。而OpenMIPS处理器具有lw、sw等加载存储指令，能够实现压栈、出栈。

综合上述分析可知， μ C/OS-II能够被移植到OpenMIPS处理器上。

从图15-2可知，移植涉及的文件既有C代码，也有汇编代码，必然涉及两者的混合编程，还涉及函数调用，所以，为了更好地理解移植过程，读者朋友应该首先阅读接下来的两节：C语言中使用汇编代码、MIPS函数调用规范。如果对这两小节的内容比较熟悉，那么可以直接阅读15.11节，进入移植过程。

15.9 C语言中使用汇编代码

内核代码的绝大部分使用C语言编写，只有一小部分使用汇编语言编写，例如：与特定体系结构相关的代码、对性能影响很大的代码。SDE MIPS编译器提供了内嵌汇编的功能，可在C代码中直接内嵌汇编语句，方便了程序设计。一个简单的内嵌汇编示例如下。

```
asm volatile ("syscall")
```

“asm”表示后面的代码为内嵌汇编；“volatile”用来告诉编译器不要优化此处的代码，以使后面的指令保留原样；括号里面是汇编指令，此处就是系统调用指令syscall。

内嵌汇编语法如下。

```
asm (汇编语句模板 : 输出部分 : 输入部分 : 破坏描述部分)
```

共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用“:”分隔开，汇编语句模板必不可少，其余三部分可选。如果某一部分没有使用，但其后面的部分使用了，那么要为其保留位置。对这四个部分分别说明如下。

1. 汇编语句模板

汇编语句模板由汇编语句序列组成，语句之间使用“;”、“/n”或“/n/t”分开。指令中的操作数可以使用占位符引用C语言变量，占位符最多10个，名称如下：%0、%1、…%9。指令中使用占位符表示的操作数，总被认为long型（4个字节），但对其施加的操作根据指令可以是半字或者字节，当把操作数当做半字或者字节使用时，默认为低半字或者LSB。对字节操作可以显式地指明是低字节还是高字节。方法是在“%”和序号之间插入一个字母，“b”代表低字节，“h”代表高字节，例如：%h2。

2. 输出部分

输出部分描述输出操作数，不同的操作数描述符之间用逗号隔开，每个操作数描述符由限定字符串和C语言变量组成。每个输出操作数的限定字符串必须包含“=”表示它是一个输出操作数。限定字符串与具体的处理器架构有关，常用限定字符串与MIPS架构可能用到的限定字符串如表15-1所示。

表15-1 内嵌汇编中常用限定字符串的描述

限定字符串	描述
d	地址寄存器
r	将操作数放入通用寄存器
f	将操作数放入浮点寄存器
g	

	操作数放在内存或通用寄存器
i	操作数是立即数
m	操作数是内存变量
o	操作数是内存变量，但是其寻址方式是偏移量类型，也即是基址寻址，或者是基址加变址寻址
I	操作数是一个16位有符号常数
J	操作数是整数0
K	操作数是一个16位无符号常数
L	操作数是一个32位有符号常数，且该常数的低16位都为0
P	操作数是1-65535之间的常数
G	操作数是浮点0
V	操作数是内存变量，但寻址方式不是偏移量类型
X	操作数可以是任何类型
=	输出操作数表达式是只写的
+	输出操作数表达式是可读可写的

amp;

输出操作数表达式独占为其指定的寄存器

3. 输入部分

输入部分描述输入操作数，不同的操作数描述符之间用逗号隔开，每个操作数描述符由限定字符串和C语言表达式或者C语言变量组成。与输出部分不同之处在于限定字符串不需要包含“=”。

4. 破坏描述部分

通常编写程序只使用一种语言：高级语言或者汇编语言。高级语言编译的步骤大致经过：预处理->编译->汇编->链接，我们这里只关心其中的编译，用来将高级语言转换成汇编代码。在转换的过程中，所有的寄存器都由编译器决定如何分配使用，编译器有能力保证寄存器的使用不会冲突。如果全部使用汇编语言，则由程序员去控制寄存器的使用，也能避免寄存器冲突。但是如果两种语言混合使用，情况就变复杂了，因为内嵌的汇编代码可以直接使用寄存器，而编译器并不去检查内嵌的汇编代码使用了哪些寄存器。因此，需要一种机制告知编译器我们使用了哪些寄存器（程序员自己知道内嵌汇编代码中使用了哪些寄存器），否则对这些寄存器的使用就有可能导致错误，“破坏描述部分”就起这个作用。破坏描述部分由逗号隔开的字符串组成，每个字符串描述一种情况，一般是寄存器名，此外还有“memory”。当然，在内嵌汇编的输入、输出部分指明的寄存器或者指定为“r”、“g”型由编译器去分配的寄存器就不需要在破坏描述部分描述，因为编译器已经知道了。

上面介绍的概念比较枯燥，还是通过例子来理解，读者只需要理解这两个例子，因为在移植μC/OS-II的过程中也只使用到了这两种形

式。

例一：asm ("mfc0 %0,\$13" :"=r"(cause_val))

指令是mfc0，读取协处理器CP0中Cause寄存器的值，保存到cause_val中，并且cause_val是通用寄存器。

例二：asm volatile("mtc0 %0,\$12" ::"r"(0x10000401))

指令是mtc0，设置协处理器CP0中Status寄存器的值为0x10000401。注意：此处只有输入部分，没有输出部分，但是要为输出部分保留位置。

15.10 MIPS函数调用规范

在移植μC/OS-II的过程中，甚至在以后的应用中，都可能使用C语言和MIPS汇编语言进行混合编程，必然涉及两者之间的相互调用，因此理解函数调用规范对移植、对程序设计都有很大的帮助。

本节的函数调用规范是以MIPS ABI o32版本为例进行讲解。ABI是一种接口定义与规范，编译系统需要使用这个规范编译和链接程序，因此ABI可以说是高级语言编写的应用程序转化为二进制可执行代码的标准。ABI涵盖了各种细节，如：数据类型的大小、布局和对齐；调用约定（控制着函数的参数如何传送以及如何接收返回值）；寄存器使用规范；系统调用的编码和一个应用如何向操作系统进行系统调用等等。MIPS ABI有三个版本：o32、n32、n64。

15.10.1 寄存器使用规范

MIPS ABI o32定义的寄存器使用规范在第1章已经给出，此处为方便读者阅读，再次给出，如表15-2所示。在本章的讲解中会经常使用该表。

表15-2 寄存器使用规范

寄存器名字	约定命名	用途
\$0	zero	总是为0
\$1	at	留作汇编器生成一些合成指令
\$2、\$3	v0、v1	用来存放子程序返回值
\$4~\$7	a0~a3	调用子程序时，使用这4个寄存器传输前4个非浮点参数
\$8~\$15	t0~t7	临时寄存器，子程序使用时可以不用存储和恢复
\$16~\$23	s0~s7	子程序寄存器变量，改变这些寄存器值的子程序必须存储旧的值并在退出前恢复，对调用程序来说值不变
\$24、\$25	t8、t9	临时寄存器，子程序使用时可以不用存储和恢复

\$26、\$27	\$k0、\$k1	由异常处理程序使用
\$28或\$gp	gp	全局指针
\$29或\$sp	sp	堆栈指针
\$30或\$fp	s8/fp	子程序可以用来做堆栈帧指针
\$31	ra	存放子程序返回地址

15.10.2 参数传递

程序的运行过程往往会涉及子函数的调用，而函数调用需要传递参数，MIPS ABI o32规范中，参数传递有两种方法：使用堆栈、使用寄存器。分别说明如下。

1. 使用堆栈传递参数

堆栈就是内存中的一段存储空间，其中可以存储局部变量、保存寄存器值、传递参数。MIPS架构的硬件没有专用的堆栈操作指令，所有的堆栈操作都是使用加载、存储指令实现的。堆栈的生长方向是从高地址到低地址，寄存器sp（即\$29，参考表15-2）指向当前栈底。

函数调用时可以使用堆栈传递参数。调用函数在堆栈上建立一个数据结构来放置参数，然后使用sp指向它，第一个参数（C源代码中最左边的函数参数）放在最低位置，每个参数至少占用一个字（32位）的空间。o32规定至少要有16字节的栈空间用于参数的传递。如图15-4所示。

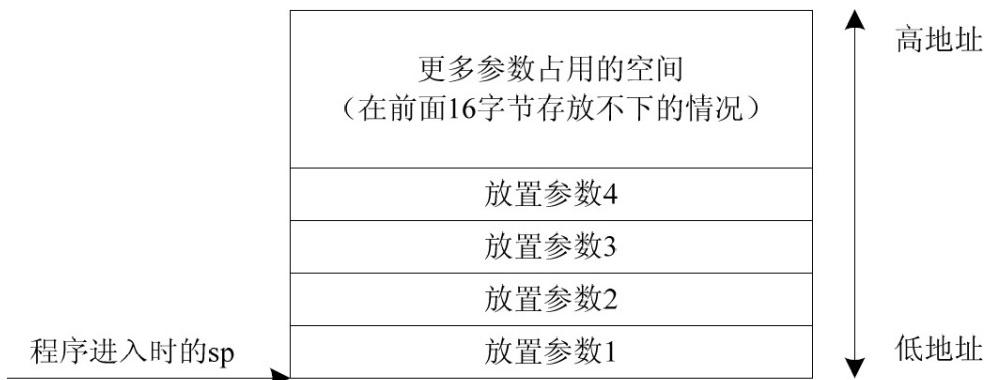


图15-4 使用堆栈传递参数

2. 使用寄存器传递参数

为了提高程序的运行效率，避免耗时的内存加载和存储操作（也就是对堆栈的操作），MIPS ABI o32规范中约定将前16个字节所对应的参数通过寄存器来传递，但同时保留堆栈上的这16个字节空间为空。即，使用寄存器传递前4个参数，但仍保留图15-4中的4个参数槽，只是不使用而已。用来传递参数的4个通用寄存器是a0-a3（即\$4-\$7，参考表15-2）。

关于MIPS ABI o32规范中的参数传递，可以总结如下：首先，无论函数参数有多少，调用函数都需要在堆栈上开辟至少16个字节的空间用于参数传递；其次，如果4个通用寄存器足够用于参数传递，那么参数就不需要存储到堆栈中开辟的16个字节的空间中，直接通过通用寄存器a0-a3传递参数，通用寄存器放不下的参数需要存储在堆栈中开辟的16个字节以上的空间中。

15.10.3 函数返回值

MIPS ABI o32规范中，整数类型或者指针类型的函数返回值会放入寄存器v0（即\$2，参考表15-2）中，而返回long long类型的数据时，也会使用寄存器v1（即\$3）。

如果返回一个结构体或其他很大的值，不能在寄存器v0、v1中完全返回，需要做如下处理：调用函数开辟一段内存缓冲区，使用一个指针指向这块缓冲区，并将这个指针作为第一个参数，通过寄存器a0，传给被调用函数。当被调用函数执行完成时，将返回值复制到这块内存缓冲区中，同时将寄存器v0指向该内存缓冲区。

15.10.4 堆栈布局

MIPS ABI o32规范中，函数的堆栈如图15-5所示（该图只是非叶子函数的堆栈）。前文已述，堆栈是从高地址向低地址生长的。

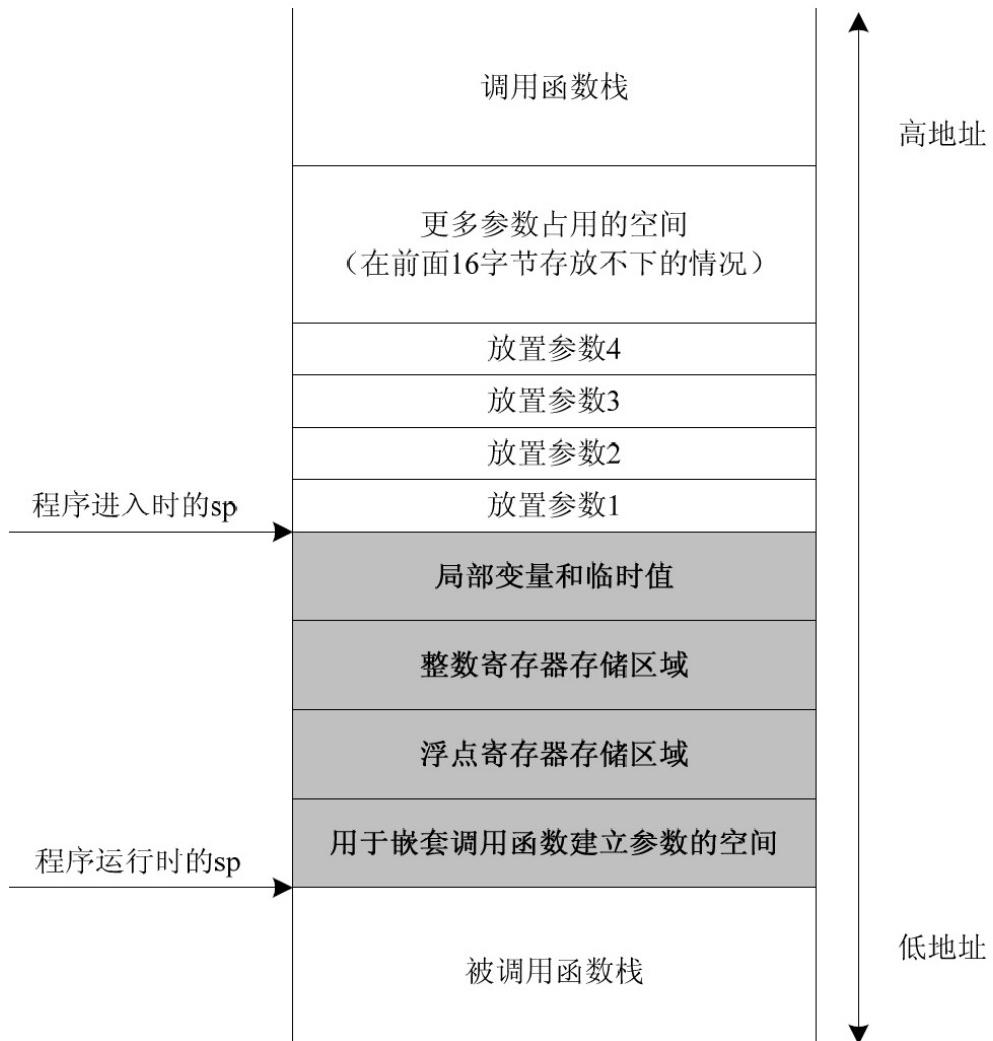


图15-5 非叶子函数的堆栈帧

图中灰色区域是函数自身需要的栈空间，其上部属于调用者。

函数分为叶子函数、非叶子函数。叶子函数是不调用其他函数的函数。它们不需要设置调用函数参数传递结构，可以安全地将数据放到寄存器t0-t7、a0-a3以及v0、v1，这些寄存器在使用之前不需要保存其中的值。

非叶子函数是会调用其他函数的函数。一般来讲，非叶子函数在运行之前，需要将寄存器ra以及其他需要事先保存的寄存器保存到堆

栈。

15.10.5 示例

通过几个例子来理解本节介绍的函数调用规范。

1. 参数传递示例

示例代码如下，作用是将传入的参数number作为GPIO模块的输出。

```
//将number作为GPIO的输出，其中GPIO_BASE是GPIO模块的基地址，在小型SOPC  
中，  
//即为0x20000000，GPIO_OUT_REG是GPIO输出寄存器的地址，为0x4。下述代码  
就是  
//将number存储到地址0x20000004，也就是设置GPIO模块的输出为number。  
  
void gpio_out(INT32U number)  
{  
    REG32(GPIO_BASE + GPIO_OUT_REG) = number;  
}
```

上述C代码在编译后，会得到如下汇编指令。将寄存器a0的值存储到地址0x20000004，根据15.10.2节介绍的参数传递可知，寄存器a0中保存的正是传递进来的参数number。

```
<gpio_out>:  
lui v0, 0x2000
```

```
ori v0,v0,0x4
sw a0,0(v0)      # 将寄存器a0的值保存到0x20000004
jr ra            # 寄存器ra保存的是返回地址，此处就是返回到调用程序
nop
```

2. 函数返回值示例

示例代码如下，作用是读取GPIO的输入。

```
//GPIO_BASE 是 GPIO 模块的基地址，在小型 SOPC 中，即为 0x20000000 ,
GPIO_IN_REG是
//GPIO输入寄存器的地址，为0x0，所以下述代码就是读取地址为0x20000000处的
值，并返回
//该值，也就是读取GPIO的输入
INT32U gpio_in()
{
    INT32U temp = 0;
    temp = REG32(GPIO_BASE + GPIO_IN_REG);
    return temp;
}
```

上述C代码在编译后，会得到如下汇编指令，主要内容是加载0x20000000处的字，保存到寄存器v0中，然后返回，正是15.10.3节介绍的函数返回规范所要求的。

```
<gpio_in>:
lui v1,0x2000
lw v0,0(v1)      # 加载0x20000000处的字，保存到寄存器v0中
```

```
jr ra          # 寄存器ra保存的是返回地址，此处就是返回到调用程序  
nop
```

3. 非叶子函数堆栈布局示例

示例代码如下，这是用户创建的一个任务，其中初始化定时器，每隔100ms通过串口输出Info数组中的两个字节，同时改变GPIO的输出。详细的解释会在15.12节编写测试程序的时候再介绍，读者此时只需要明白一点——这个函数是非叶子函数，因为在其中会调用OSInitTick、uart_putc、gpio_out、OSTimeDly等函数。我们需要考察的是其堆栈布局。

```
void TaskStart (void *pdata)  
{  
    INT32U count = 0;  
    pdata = pdata;           /* 没有作用，仅仅是防止编译器给出警告信息  
 */  
    OSInitTick();           /* 初始 化 定 时 器  
 */  
    for (;;) {  
        if(count <= 102)  
        {  
            uart_putc(Info[count]);      /* 通 过 串 口 输出 Info 中 的 两 个 字 节  
 */  
            uart_putc(Info[count+1]);  
        }  
        gpio_out(count);             /* 改 变 GPIO 的 输 出  
 */  
    }  
}
```

```
    count=count+2;  
    OSTimeDly(10); /* 等待 100ms  
*/  
}  
}
```

上述C代码在编译后，会得到如下汇编指令（只截取前几条指令）。

```
<TaskStart>:  
addiu sp, sp, -48      # 将堆栈指针sp减去48  
sw ra, 44(sp)          # 将寄存器ra、s6、s5、s4、.....s0依次压栈  
sw s6, 40(sp)  
sw s5, 36(sp)  
sw s4, 32(sp)  
sw s3, 28(sp)  
sw s2, 24(sp)  
sw s1, 20(sp)  
sw s0, 16(sp)  
.....
```

首先将堆栈指针sp减去48，也就是向低地址方向移动12个字。然后将寄存器ra、s6、s5、s4、...s0依次压入堆栈。如图15-6所示。这样在函数TaskStart内部就可以自由使用寄存器ra、s6、s5、s4、...s0了。

4. 叶子函数堆栈布局示例

上面给出的参数传递示例、函数返回值示例都是叶子函数，所以此处不再单独给出叶子函数示例。从那两个示例对应的汇编代码可以发现，叶子函数不需要堆栈（实际上，根据需要也可以拥有堆栈），没有要保存的寄存器。

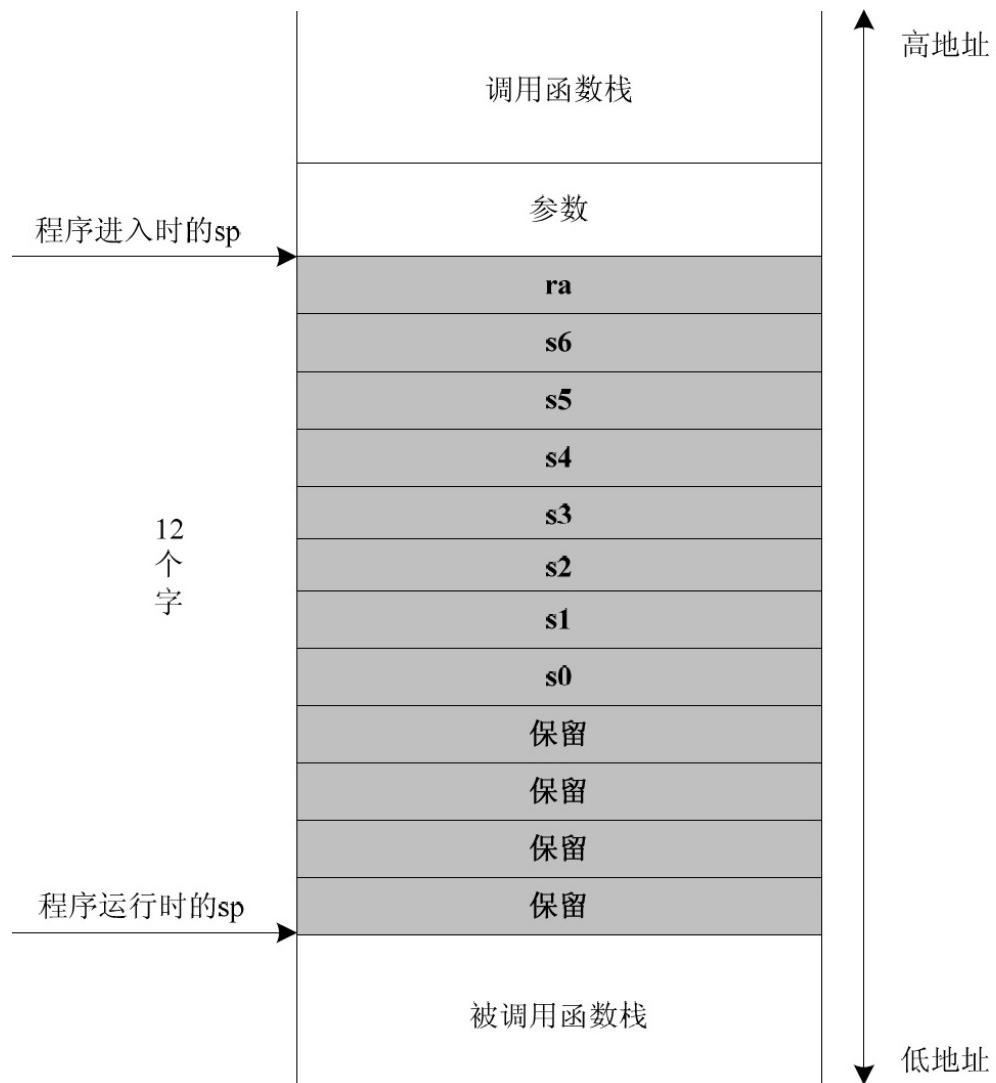


图15-6 TaskStart函数运行时的堆栈

15.11 μC/OS-II在OpenMIPS处理器上的移植

有了前几节介绍的背景知识，现在就可以开始移植μC/OS-II到OpenMIPS处理器了。

根据目标处理器的不同，移植μC/OS-II需要修改50-300行代码。最方便的途径是修改现有的移植代码，比如本章要将μC/OS-II移植到OpenMIPS处理器，那么可以借鉴μC/OS-II在其它MIPS架构处理器上的移植代码，在其基础上进行修改。

15.11.1 文件目录的建立

本小节将建立代码文件的目录，分以下步骤。

1. 在Ubuntu虚拟机中新建文件夹ucosii_OpenMIPS，作为整个移植过程的根目录。
2. 从<http://micrium.com/>下载μC/OS-II的源代码，版本是v2.91。在本书附带光盘Code\ucosii_sourcecode目录下也提供了全部源代码，文件名是uCOS-II-V290.ZIP。解压缩后，包含的代码文件如图15-7所示。
3. 在ucosii_OpenMIPS目录下新建文件夹ucos，将μC/OS-II源代码的文件（除了os_cfg_r.h、ucos_ii.h两个头文件之外）复制到ucos文件夹下。



图15-7 μC/OS-II源代码提供的文件

4. 从<http://micrium.com/>下载针对MIPS M14K的μC/OS-II移植代码，版本是v2.90。本书附带光盘的Code\ucosii_sourcecode目录下也提供了移植代码，文件名是uCOS-II_M14K.zip。MIPS M14K是MIPS科技于2009年发布的首款执行microMIPS指令集架构的MIPS32兼容内核。microMIPS指令集架构集成了16位和32位优化指令的高性能代码压缩技术，保持了98%的MIPS32性能，同时减少了至少30%的代码体积，从而降低芯片成本，也有助于降低系统功耗。本章的移植工作就是在针对MIPS M14K的μC/OS-II移植代码的基础上修改完成的。移植代码的文件如图15-8所示。注意：这几个文件并不在同一个文件夹下，此处只是为了显示方便而将其复制到同一个文件夹下。



图15-8 针对MIPS M14K的μC/OS-II移植代码

5. 在ucosii_OpenMIPS目录下新建文件夹port，将针对MIPS M14K的μC/OS-II移植代码中的os_cpu_a.S、os_cpu_c.c两个文件复制到该文件夹下。

6. 在ucosii_OpenMIPS目录下新建文件夹include，将μC/OS-II源代码中的ucos_ii.h、os_cfg_r.h两个头文件，以及针对MIPS M14K的μC/OS-II移植代码中的cpu.h、os_cpu.h两个头文件，一共四个头文件复制到该文件夹下，并将os_cfg_r.h重命名为os_cfg.h，注意：在重命名前需要去掉该文件的只读属性，否则无法重命名。

7. 在include目录下新建文件includes.h，内容如下。

```
#include <stdarg.h>
#include <stddef.h>
#include <limits.h>
#include "ucos_ii.h"
```

主要是因为针对MIPS M14K的μC/OS-II移植代码中的os_cpu_c.c文件需要引用includes.h文件，所以此处添加该文件。

8. 在include目录下新建文件app_cfg.h，内容如下。

```
#ifndef _APP_CFG_H_
#define _APP_CFG_H_

#define OS_TASK_TMR_PRIO (OS_LOWEST_PRIO - 2)

#endif
```

从V2.81版本开始，μC/OS-II增加了周期性（Periodic）和一次性（One-Shot）的定时器功能。用户程序最多可以使用65500个定时器，这些定时器由一个定时器管理任务进行管理，该任务的优先级就定义在刚刚新建的app_cfg.h文件中，即OS_TASK_TMR_PRIO。

9. 修改includes目录下的cpu.h文件，去掉其中对如下两个头文件的引用，因为移植过程没有用到这两个头文件。

```
/* 去掉如下代码： */
#include <cpu_def.h>
#include <cpu_cfg.h>
```

10. 在ucosii_OpenMIPS目录下新建common文件夹，其中用于存放测试程序。

经过上述步骤，就建立了代码文件的目录，如图15-9所示。接下来需要修改部分代码，主要是修改port目录下的文件以及os_cpu.h文件，以实现移植μC/OS-II到OpenMIPS处理器。由于本章的移植是建立在针对MIPS M14K的移植代码基础之上的，所以不会有大的改动，只有一些细微的修改，但是如果笔者只介绍这些细微的修改，读者难免会有盲人摸象的感觉，对移植需要做的工作、移植的原理还是不清楚，所以，笔者决定详细介绍port中的每个文件以及os_cpu.h文件，以帮助读者理解。

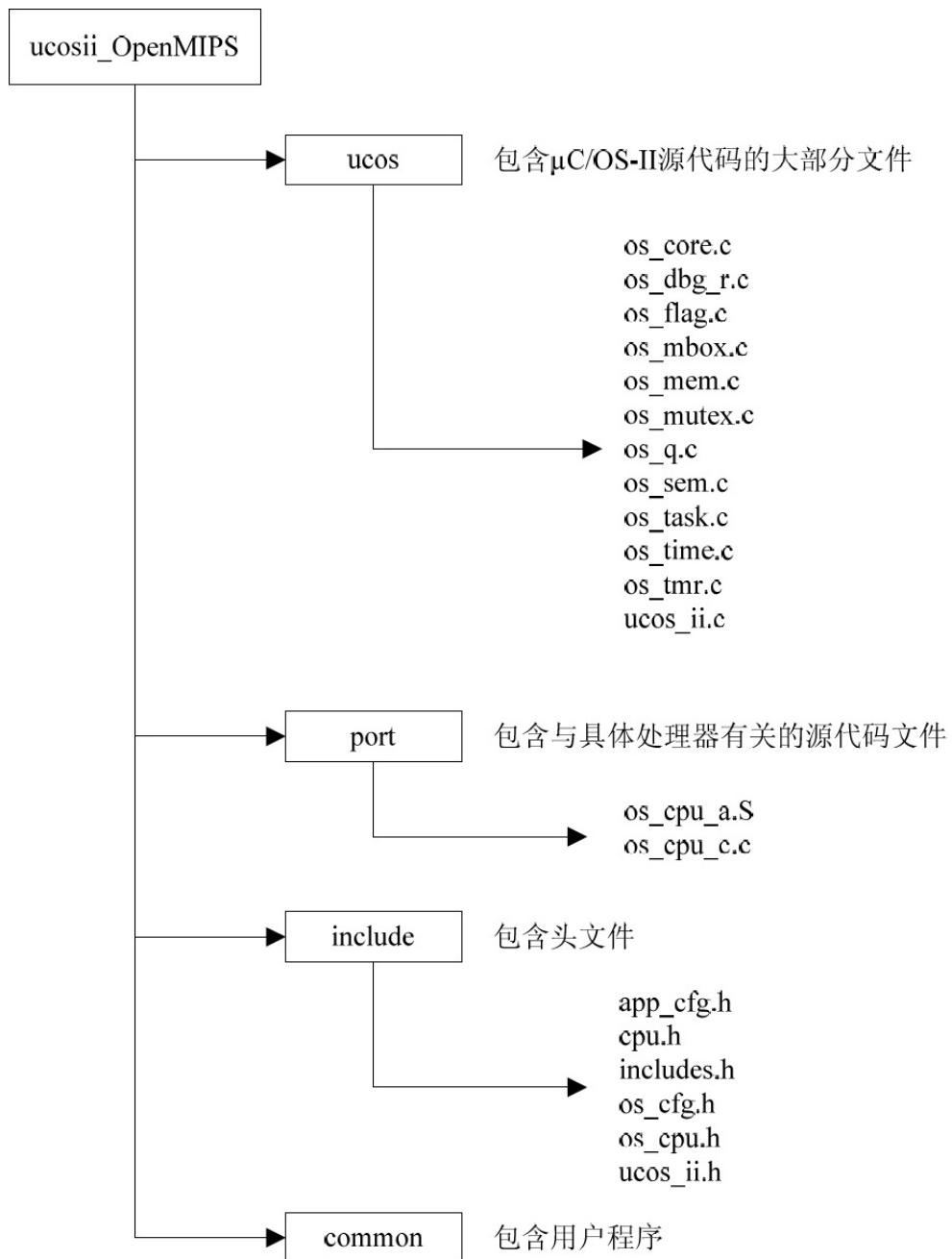


图15-9 代码文件的目录

15.11.2 修改os_cpu.h文件

os_cpu.h文件定义了与处理器相关的常量、宏、结构体。主要内容如下，完全可以使用针对MIPS M14K的移植代码，不用修改。

1. 数据类型定义

因为不同的处理器有不同的字长，所以μC/OS-II包含了一系列的数据类型定义，以确保其可移植性。尤其是，μC/OS-II代码从不使用C语言中的short、int、long等数据类型，因为它们是与编译器相关的，是不可移植的。μC/OS-II使用自己定义的数据类型，如下所示。

```
*****          数据类型定义，与编译器有关
*****
typedef unsigned char      BOOLEAN;
typedef unsigned char      INT8U; /* 无符号8位整数 */
typedef signed   char      INT8S; /* 有符号8位整数 */
typedef unsigned short     INT16U; /* 无符号16位整数 */
typedef signed   short     INT16S; /* 有符号16位整数 */
typedef unsigned int       INT32U; /* 无符号32位整数 */
typedef signed   int       INT32S; /* 有符号32位整数 */
typedef float             FP32;
typedef double            FP64;

typedef unsigned int      OS_STK;      /* 堆栈宽度,
是32位 */
typedef unsigned int      volatile    OS_CPU_SR;
```

比如，如果某一处理器的编译器认为int是有符号16位整数，而不是short，那么只需要将INT16S前面的的short改为int即可，不用修改

μ C/OS-II的其余代码，以此确保可移植性。

2. 进、出临界区的宏

如果某操作不希望被打断，那么可将该操作放入临界区中， μ C/OS-II在进入临界区之前要禁止中断，操作完毕后，退出临界区需要开中断。 μ C/OS-II定义了两个宏来进入临界区、退出临界区，其中就分别实现禁止中断、开中断。这两个宏的定义如下。

```
/*****                                     进、出临界区的宏
*****
#define OS_ENTER_CRITICAL()    cpu_sr = OS_CPU_SR_Save();
#define OS_EXIT_CRITICAL()     OS_CPU_SR_Restore(cpu_sr);
```

其中OS_CPU_SR_Save、OS_CPU_SR_Restore两个函数在文件os_cpu_a.S中定义，会在15.11.3节介绍这两个函数。

有了这两个宏，执行临界区代码的过程如下。

```
OS_CPU_SR cpu_sr;
.....
OS_ENTER_CRITICAL()
```

临界区代码

```
OS_EXIT_CRITICAL()
```

.....

3. 定义堆栈生长方向

大多数处理器的堆栈是从高地址向低地址生长的，但也有部分处理器相反。 μ C/OS-II被设计成对这两种情况都可以处理，只要配置如下的宏定义即可。

```
#define OS_STK_GROWTH 1
```

当OS_STK_GROWTH等于1时，表示堆栈的生长方向是从高地址向低地址，当OS_STK_GROWTH等于0时，表示堆栈的生长方向是从低地址向高地址。对OpenMIPS处理器而言，此处设置为1。

4. 用于任务切换的宏定义

```
#define OS_TASK_SW() asm("\tsyscall\n");
```

宏定义OS_TASK_SW()实现了任务切换，是在 μ C/OS-II从低优先级任务切换到高优先级任务时需用到的。从定义中可以发现该宏定义实际就是系统调用指令syscall。

假如任务TaskA在执行过程中，由于某种原因需要执行任务TaskB，且TaskB的优先级高于TaskA，那么可以在TaskA中执行任务切换宏OS_TASK_SW()，后者会引发系统调用异常，进入异常处理例程。在异常处理例程的最后， μ C/OS-II会查找当前优先级最高的就绪任务开始执行，于是就会执行TaskB，从而实现了低优先级任务向高优先级任务的切换。

5. 一些函数声明

os_cpu.h 文件的最后给出了一些函数声明，这些函数在文件 os_cpu_a.S 中定义，15.11.3 小节会详细说明。

```
void      OSIntCtxSw(void);
void      OSStartHighRdy(void);
void      ExceptionHandler(void);
void      InterruptHandler(void);

void      TickInterruptClear(void);
void      CoreTmrInit(CPU_INT32U tmr_reload);
void      TickISR(CPU_INT32U tmr_reload);

OS_CPU_SR  OS_CPU_SR_Save(void);
void      OS_CPU_SR_Restore(OS_CPU_SR);
```

15.11.3 修改os_cpu_a.S文件

os_cpu_a.S 文件定义了异常处理例程，此外还定义了一些常数、11 个函数，如下。

- OS_CPU_SR_Save
- OS_CPU_SR_Restore
- InterruptHandler
- OSIntCtxSw
- ExceptionHandler

- OSStartHighRdy
- TickInterruptClear
- CoreTmrInit
- TickISR
- DisableInterruptSource
- EnableInterruptSource

分别介绍如下。

1. 异常处理例程

其中定义了stack段、定义了异常处理例程。代码如下。

```

/***** 定义了 stack 段 *****/
***** /
    .section .stack, "aw", @nobits
    .space 0x10000

/***** 定义了 vectors 段， 其中存放异常处理例程 ****/
***** /
    .section .vectors, "ax"

/***** 复位异常， 对应的入口地址是 0x0 ****/
***** /
    .org 0x0

_reset:
    lui $28,0x0      /* 寄存器$28即全局指针寄存器gp */
    la $29,_stack_addr /* 寄存器$29即堆栈指针寄存器sp */

```

```
la $26,main           /* 寄存器$26、$27留给异常处理程序使用 */

jr $26

nop

/***** 中断异常，对应的入口地址是 0x20 *****/
.org 0x20

la $26,InterruptHandler
jr $26
nop

***** 系统调用异常、无效指令、溢出异常、自陷异常，对应的入口地址是0x40 ****/
.org 0x40

la $26,ExceptionHandler
jr $26
nop
```

上述代码很好理解。首先定义了堆栈段，大小是0x10000字节。然后定义了vectors段，其中包括三个异常处理例程。

(1) 复位异常，对应的处理例程入口地址是0x0。在其中初始化全局指针寄存器gp、堆栈指针寄存器sp，然后转移到main函数。其中将_stack_addr的值作为sp的初始值，_stack_addr是在链接指示文件ram.ld中定义的（参考15.13节），对应的是堆栈的最高地址。

(2) 中断异常，对应的处理例程入口地址是0x20。在其中直接转移到函数InterruptHandler进行中断处理。读者可能会有疑问，一般而

言，异常处理之前应该先保护现场，即保存所有寄存器的值，此处在没有保护现场的情况下就使用了寄存器\$26，会不会出问题？保护现场的工作在函数InterruptHandler中进行，此处在没有保护现场的情况下就使用了寄存器\$26，是因为寄存器\$26（以及寄存器\$27）就是给异常处理程序使用的，其它程序不使用，所以修改了也没关系，参考表15-2。

(3) 系统调用异常、无效指令、溢出异常、自陷异常，对应的处理例程入口地址是0x40。在其中直接转移到函数ExceptionHandler进行异常处理。此处也使用了寄存器\$26。

需要注意的是，在代码中使用了la指令，这是一个汇编指令，是与编译器有关的指令，用来将指定的地址加载到寄存器，其等价于如下两条机器指令。

```
//la指令用来将指定的地址加载到寄存器，等价于两条机器指令，如下，  
//其中%hi(addr)表示addr的高16bit，%lo(addr)表示addr的低16bit  
la rt, addr    =>    lui    $2, %hi(addr)  
                      addiu $2, $2, %lo(addr)
```

2. 一些常数

os_cpu_a.S定义了出入堆栈使用到的一些常数定义。

```
.equ STK_OFFSET_SR,        4  
.equ STK_OFFSET_EPC,      STK_OFFSET_SR + 4  
.equ STK_OFFSET_L0,       STK_OFFSET_EPC + 4  
.equ STK_OFFSET_HI,       STK_OFFSET_L0 + 4  
.equ STK_OFFSET_GPR1,     STK_OFFSET_HI + 4
```

```
.equ STK_OFFSET_GPR2, STK_OFFSET_GPR1 + 4
.equ STK_OFFSET_GPR3, STK_OFFSET_GPR2 + 4
.equ STK_OFFSET_GPR4, STK_OFFSET_GPR3 + 4
.equ STK_OFFSET_GPR5, STK_OFFSET_GPR4 + 4
.equ STK_OFFSET_GPR6, STK_OFFSET_GPR5 + 4
.equ STK_OFFSET_GPR7, STK_OFFSET_GPR6 + 4
.equ STK_OFFSET_GPR8, STK_OFFSET_GPR7 + 4
.equ STK_OFFSET_GPR9, STK_OFFSET_GPR8 + 4
.equ STK_OFFSET_GPR10, STK_OFFSET_GPR9 + 4
.equ STK_OFFSET_GPR11, STK_OFFSET_GPR10 + 4
.equ STK_OFFSET_GPR12, STK_OFFSET_GPR11 + 4
.equ STK_OFFSET_GPR13, STK_OFFSET_GPR12 + 4
.equ STK_OFFSET_GPR14, STK_OFFSET_GPR13 + 4
.equ STK_OFFSET_GPR15, STK_OFFSET_GPR14 + 4
.equ STK_OFFSET_GPR16, STK_OFFSET_GPR15 + 4
.equ STK_OFFSET_GPR17, STK_OFFSET_GPR16 + 4
.equ STK_OFFSET_GPR18, STK_OFFSET_GPR17 + 4
.equ STK_OFFSET_GPR19, STK_OFFSET_GPR18 + 4
.equ STK_OFFSET_GPR20, STK_OFFSET_GPR19 + 4
.equ STK_OFFSET_GPR21, STK_OFFSET_GPR20 + 4
.equ STK_OFFSET_GPR22, STK_OFFSET_GPR21 + 4
.equ STK_OFFSET_GPR23, STK_OFFSET_GPR22 + 4
.equ STK_OFFSET_GPR24, STK_OFFSET_GPR23 + 4
.equ STK_OFFSET_GPR25, STK_OFFSET_GPR24 + 4
.equ STK_OFFSET_GPR26, STK_OFFSET_GPR25 + 4
.equ STK_OFFSET_GPR27, STK_OFFSET_GPR26 + 4
.equ STK_OFFSET_GPR28, STK_OFFSET_GPR27 + 4
```

```

.equ      STK_OFFSET_GPR30,      STK_OFFSET_GPR28 + 4
.equ      STK_OFFSET_GPR31,      STK_OFFSET_GPR30 + 4
.equ      STK_CTX_SIZE,          STK_OFFSET_GPR31 + 4 /* 堆栈的大小
*/

```

从常数的名称上可以分析，这些常数定义了对应的寄存器在堆栈中的位置（相对于sp的偏移），如图15-10所示。当异常发生时，会按照图15-10的次序将各个寄存器保存到堆栈，在后文解释函数InterruptHandler时会详述。另外，上述定义中的STK_CTX_SIZE表示堆栈的大小。

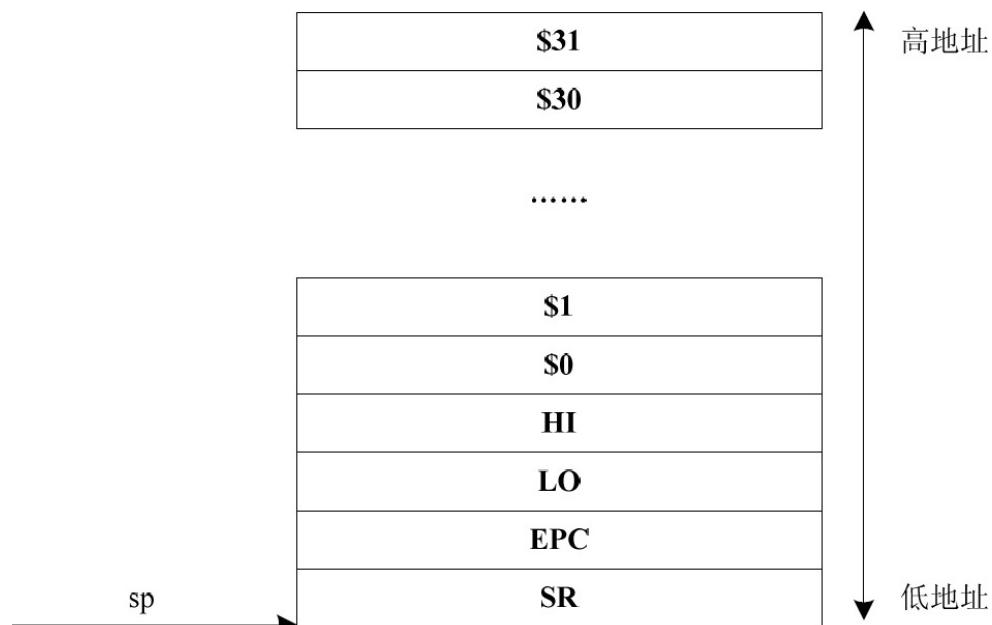


图15-10 寄存器在堆栈中的保存位置

3. 函数OS_CPU_SR_Save

作用是读取协处理器CP0中Status寄存器的值，同时禁止中断，函数声明如下。

```
CPU_SR OS_CPU_SR_Save(void);
```

该函数有一个返回值，返回的是CP0中Status寄存器的值。函数定义如下。

```
.ent OS_CPU_SR_Save  
OS_CPU_SR_Save:  
  
    ori    $2,$2,0x0  
    mfc0  $2,$12,0          # 获取Status寄存器的值，保存到寄存器$2中  
    addi   $3,$0,0xffffe     # 设置寄存器$3的值为0xffffffff  
    and    $3,$2,$3          # $3与$2的值相与，结果保存到$3中  
    mtc0  $3,$12,0          # 修改后的寄存器$3的值保存到Status寄存器  
    jr    $31                 # 返回  
    nop  
  
.end OS_CPU_SR_Save
```

首先获取当前Status寄存器的值，保存到寄存器v0（即程序中的\$2）中，然后修改其最低位为0，修改后的值保存到v1（即程序中的\$3）中，再将v1的值保存到Status寄存器，由于v1的最低位为0，从而禁止中断（参考10.2节对Status寄存器的说明）。最后返回。

根据MIPS函数调用规范可知，函数返回值一般放在寄存器v0中，而此时v0的值正是修改之前Status寄存器的值。满足规范要求，达到了本函数的目的。

4. 函数OS_CPU_SR_Restore

作用是恢复Status寄存器的值，函数声明如下，从中可以发现该函数有一个传入参数。

```
void OS_CPU_SR_Restore(CPU_SR sr);
```

函数定义如下。

```
.ent OS_CPU_SR_Restore
```

```
OS_CPU_SR_Restore:  
  
    jr      $31  
    mtc0   $4, $12, 0          //参数是通过寄存器a0（即$4）传入的  
  
.end OS_CPU_SR_Restore
```

根据MIPS函数调用规范可知，参数是通过寄存器a0-a3（即\$4-\$7）传递的，所以上述代码很好理解，将传入参数存储到协处理器CP0中的Status寄存器。需要注意的是，上述代码将mtc0指令放在延迟槽中。

通过分析实际的函数，读者朋友应该对15.10节介绍的MIPS函数调用规范有了更加深刻的理解了吧。

5. 函数InterruptHandler

在前文定义的中断处理例程中，会直接转移到函数InterruptHandler，进行中断处理，函数InterruptHandler的定义如下。

```
.ent InterruptHandler
```

```
InterruptHandler:
```

```
/* **** */
**      第一段：保护现场，寄存器压栈      ****
***** */
*/
```

```
addi $29, $29, -STK_CTX_SIZE /* 调整堆栈指针 */
```

```
sw $1, STK_OFFSET_GPR1($29) /* 保存整数寄存器 */
```

```
sw $2, STK_OFFSET_GPR2($29)
```

```
sw $3, STK_OFFSET_GPR3($29)
```

```
sw $4, STK_OFFSET_GPR4($29)
```

```
sw $5, STK_OFFSET_GPR5($29)
```

```
sw $6, STK_OFFSET_GPR6($29)
```

```
sw $7, STK_OFFSET_GPR7($29)
```

```
sw $8, STK_OFFSET_GPR8($29)
```

```
sw $9, STK_OFFSET_GPR9($29)
```

```
sw $10, STK_OFFSET_GPR10($29)
```

```
sw $11, STK_OFFSET_GPR11($29)
```

```
sw $12, STK_OFFSET_GPR12($29)
```

```
sw $13, STK_OFFSET_GPR13($29)
```

```
sw $14, STK_OFFSET_GPR14($29)
```

```
sw $15, STK_OFFSET_GPR15($29)
```

```
sw $16, STK_OFFSET_GPR16($29)
```

```
SW      $17, STK_OFFSET_GPR17($29)
SW      $18, STK_OFFSET_GPR18($29)
SW      $19, STK_OFFSET_GPR19($29)
SW      $20, STK_OFFSET_GPR20($29)
SW      $21, STK_OFFSET_GPR21($29)
SW      $22, STK_OFFSET_GPR22($29)
SW      $23, STK_OFFSET_GPR23($29)
SW      $24, STK_OFFSET_GPR24($29)
SW      $25, STK_OFFSET_GPR25($29)
SW      $26, STK_OFFSET_GPR26($29)
SW      $27, STK_OFFSET_GPR27($29)
SW      $28, STK_OFFSET_GPR28($29)
SW      $30, STK_OFFSET_GPR30($29)
SW      $31, STK_OFFSET_GPR31($29)

mflo  $8
mfhi  $9
SW      $8, STK_OFFSET_LO($29)      /* 保存寄存器LO */
SW      $9, STK_OFFSET_HI($29)      /* 保存寄存器HI */

mfc0  $8, $14, 0
SW      $8, STK_OFFSET_EPC($29)      /* 保存寄存器EPC */

mfc0  $8, $12, 0
SW      $8, STK_OFFSET_SR($29)      /* 保存寄存器SR，也就是
Status寄存器 */
```

```
*****  
**  
***** 第二段：变量OSIntNesting加1 *****  
*****  
**/  
  
la    $8,  OSIntNesting  
lbu   $9,  0($8)          /* 获取OSIntNesting的值  
*/  
  
bne   $0,  $9,  TICK_INC_NESTING  /* OSIntNesting不为零，则转  
移到 */  
/* TICK_INC_NESTING           */  
nop  
  
/* OSIntNesting为零，则进  
行下面的操作 */  
  
la    $10,  OSTCBCur  
lw    $11,  0($10)  
sw    $29,  0($11)          /* 将当前任务的堆栈指针保存到任务控制块  
OSTCBCur中 */  
  
TICK_INC_NESTING:  
  
addi  $9,  $9,  1          /* OSIntNesting的值加1 */  
sb    $9,  0($8)
```

```
/*
**
***** 第三段：中断处理 *****
*/
INT_LOOP:

    mfco $8, $13          /* 读取Cause寄存器 */
    li    $9, 0xff00        /* 使$9为0x0000ff00 */
    and   $8, $9            /* 获取Cause寄存器中的IP字
段 */
    clz   $8, $8            /* 通过IP字段判断是否有中断
发生 */
    move  $4, $8
    la    $8, BSP_Interrupt_Handler

    li    $9, 32
    beq  $4, $9, INT_LOOP_END      /* 中断个数为0，那么转到
INT_LOOP_END */

    nop

    jalr $8                /* 否则，进入具体的中断处理函
数 */
    /* BSP_Interrupt_Handler */
*/
```

```
nop

    b      INT_LOOP          /* 处理结束一个中断后，继续处理
其余中断 */

nop

/***** 第四段：中断处理结束 *****
**                         第四段：中断处理结束                         **
***** */

INT_LOOP_END:

    la      $8,  OSIntExit          /* 中断处理结束后，调用函数
OSIntExit */
    jalr   $8
    nop

/***** 第五段：恢复现场 *****
**                         第五段：恢复现场                         **
***** */

    lw      $8,  STK_OFFSET_SR($29)
    mtc0  $8,  $12,  0           /* 恢复寄存器SR */
```

```
lw      $8,  STK_OFFSET_EPC($29)
mtc0  $8,  $14,  0           /* 恢复寄存器EPC */

lw      $8,  STK_OFFSET_LO($29)
lw      $9,  STK_OFFSET_HI($29)
mtlo  $8           /* 恢复寄存器LO */
mtlo  $9           /* 恢复寄存器HI */

lw      $31,  STK_OFFSET_GPR31($29) /* 恢复整数寄存器 */
lw      $30,  STK_OFFSET_GPR30($29)
lw      $28,  STK_OFFSET_GPR28($29)
lw      $27,  STK_OFFSET_GPR27($29)
lw      $26,  STK_OFFSET_GPR26($29)
lw      $25,  STK_OFFSET_GPR25($29)
lw      $24,  STK_OFFSET_GPR24($29)
lw      $23,  STK_OFFSET_GPR23($29)
lw      $22,  STK_OFFSET_GPR22($29)
lw      $21,  STK_OFFSET_GPR21($29)
lw      $20,  STK_OFFSET_GPR20($29)
lw      $19,  STK_OFFSET_GPR19($29)
lw      $18,  STK_OFFSET_GPR18($29)
lw      $17,  STK_OFFSET_GPR17($29)
lw      $16,  STK_OFFSET_GPR16($29)
lw      $15,  STK_OFFSET_GPR15($29)
lw      $14,  STK_OFFSET_GPR14($29)
lw      $13,  STK_OFFSET_GPR13($29)
```

```
lw    $12, STK_OFFSET_GPR12($29)
lw    $11, STK_OFFSET_GPR11($29)
lw    $10, STK_OFFSET_GPR10($29)
lw    $9,  STK_OFFSET_GPR9($29)
lw    $8,  STK_OFFSET_GPR8($29)
lw    $7,  STK_OFFSET_GPR7($29)
lw    $6,  STK_OFFSET_GPR6($29)
lw    $5,  STK_OFFSET_GPR5($29)
lw    $4,  STK_OFFSET_GPR4($29)
lw    $3,  STK_OFFSET_GPR3($29)
lw    $2,  STK_OFFSET_GPR2($29)
lw    $1,  STK_OFFSET_GPR1($29)

addi $29, $29, STK_CTX_SIZE      /* 调整堆栈指针 */

/*
*****
* 第六段：返回
*****
*/
eret      /* 返回 */

.end InterruptHandler
```

注意其中使用到的汇编指令li，这是与编译器有关的指令，用来将立即数加载到寄存器，等价于ori指令，如下。

```
li rt, immediate 等价于 ori rt,$0,immediate
```

另外，其中部分and指令采用了简化方式，如下，编译器可以识别。在后面还会遇到or指令，也采用了类似的简化方式。

```
and rd,rs 等价于 and rd,rd,rs
```

上述代码可以分为六段理解。

第一段：保护现场，也就是将各个寄存器的值保存到堆栈，保存的格式就是图15-10所示。注意：不用保存堆栈指针寄存器sp（即\$29）。

第二段：首先获取变量OSIntNesting的值，然后分两种情况。

- 如果为零，那么需要将被中断任务的堆栈指针保存到该任务的任务控制块（即OSTCBCur）中。参考15.5.1节给出的任务控制块的结构体定义，其第一个元素就是堆栈指针，所以此处直接将寄存器sp保存到OSTCBCur指向的地址，该地址对应就是任务控制块中的堆栈指针这个元素。然后将变量OSIntNesting加1。
- 如果不为零，那么直接将变量OSIntNesting加1。

μ C/OS-II需要知道正在做中断服务，这样只要变量OSIntNesting不为0，就表示处于中断处理过程中。

第三段：中断处理过程。读取Cause寄存器的值，获得其中的IP字段，该字段位于Cause寄存器的第8-15bit。然后判断IP字段中是否有1，有1就表示有中断发生，那么会进入具体的中断处理函数BSP_Interrupt_Handler进行处理，该函数在os_cpu_c.c中定义，会在15.11.4节详细说明。中断处理结束后，还要再次判断是否有中断发生，如果有中断，那么再次进入函数BSP_Interrupt_Handler进行处理，如此反复，直到没有中断为止。

第四段：中断处理结束，此时需要调用函数OSIntExit。OSIntExit会将变量OSIntNesting减1，当OSIntNesting减到0时，表示所有嵌套的中断都处理结束。此时，μC/OS-II需要判断是否有更高优先级的任务进入就绪态（中断处理过程可能会唤醒更高优先级的任务）。如果有，那么就调用函数OSIntCtxSw以实现切换到优先级更高的任务继续执行，反之，回到原来被中断的任务。其过程如图15-11所示。

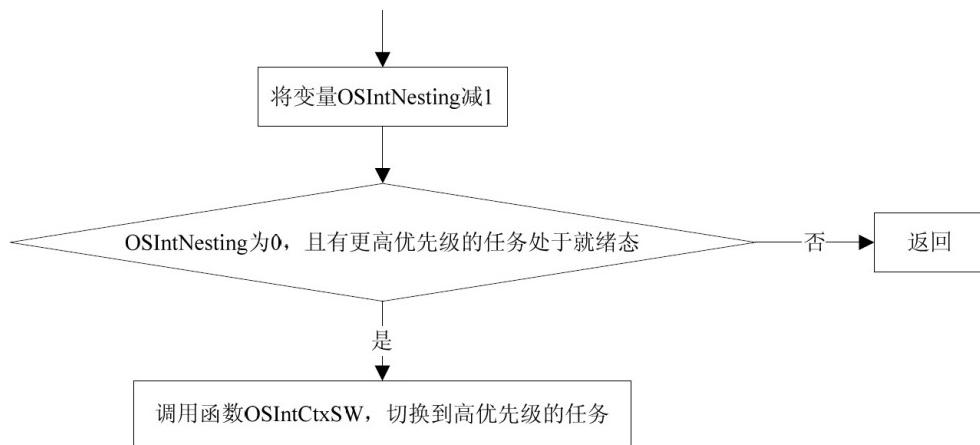


图15-11 函数OSIntExit的处理流程

函数OSIntExit是在文件os_core.c中定义的，无需修改。图15-11中的函数OSIntCtxSw是在文件os_cpu_a.S中定义的，本节接下来会介绍该函数。

第五段：从堆栈中恢复各个寄存器的值。注意：不用恢复寄存器sp（即\$29）。

第六段：调用指令eret实现返回。指令eret会将寄存器EPC的值赋给取指寄存器PC，作为新的取指地址。

再进一步思考，如果第四段中，满足“OSIntNesting为0，且有更高优先级的任务处于就绪态”这个条件，那么会切换到更高优先级的任务继续执行，而当前被中断的任务就会进入就绪态，该任务的堆栈指针sp存放在其任务控制块中，在堆栈中保存了所有寄存器的值，如图15-12所示。实际上，这就是处于就绪态的任务的普遍状态，有了这个认识，就比较容易理解下面将要解释的函数OSIntCtxSw。

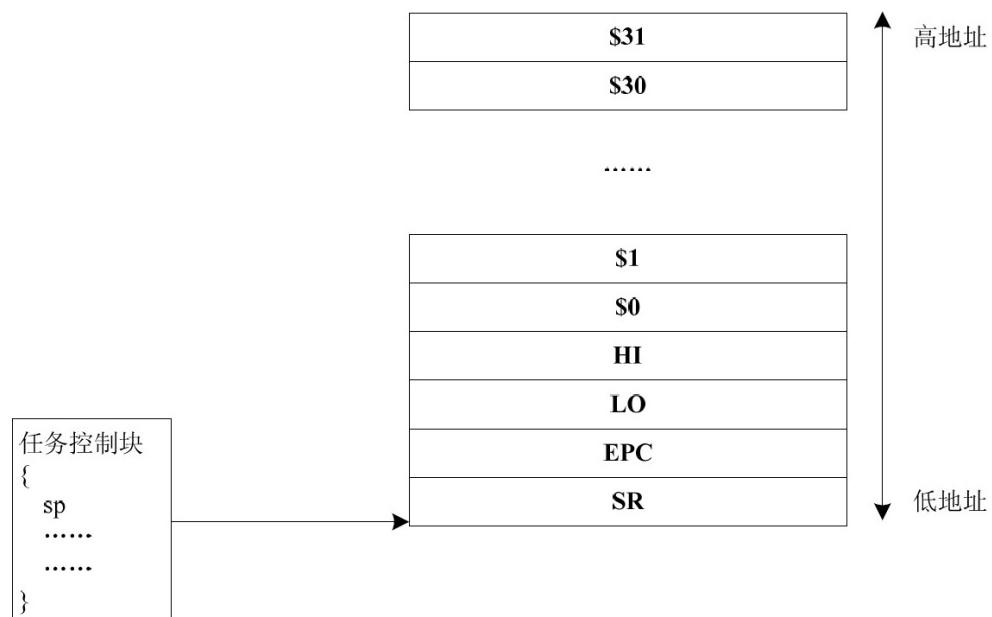


图15-12 就绪态任务的普遍状态

6. 函数OSIntCtxSw

函数OSIntCtxSw的作用在图15-11中已经提到，简单来说，就是切换到高优先级的任务继续执行。代码如下所示。


```
la    $8,  OSTCBHighRdy
lw    $9,  0($8)
la    $10, OSTCBCur
sw    $9,  0($10)           /* OSTCBCur指向当前任务的TCB，此处
就是将 */
                                         /* OSTCBCur指向最高优先级任务的TCB */

/******************* 第三段：恢复现场 *****************/
**/                                             /* **** */

lw    $29,  0($9)           /* 加载最高优先级任务的堆栈指针
*/ 

lw    $8,  STK_OFFSET_SR($29)
mtc0 $8,  $12, 0           /* 恢复寄存器SR */

lw    $8,  STK_OFFSET_EPC($29)
mtc0 $8,  $14, 0           /* 恢复寄存器EPC */

lw    $8,  STK_OFFSET_L0($29)
lw    $9,  STK_OFFSET_HI($29)
mtlo $8                   /* 恢复寄存器LO */
mthi $9                   /* 恢复寄存器HI */
```

```
lw    $31, STK_OFFSET_GPR31($29) /* 恢复整数寄存器 */

lw    $30, STK_OFFSET_GPR30($29)

lw    $28, STK_OFFSET_GPR28($29)

lw    $27, STK_OFFSET_GPR27($29)

lw    $26, STK_OFFSET_GPR26($29)

lw    $25, STK_OFFSET_GPR25($29)

lw    $24, STK_OFFSET_GPR24($29)

lw    $23, STK_OFFSET_GPR23($29)

lw    $22, STK_OFFSET_GPR22($29)

lw    $21, STK_OFFSET_GPR21($29)

lw    $20, STK_OFFSET_GPR20($29)

lw    $19, STK_OFFSET_GPR19($29)

lw    $18, STK_OFFSET_GPR18($29)

lw    $17, STK_OFFSET_GPR17($29)

lw    $16, STK_OFFSET_GPR16($29)

lw    $15, STK_OFFSET_GPR15($29)

lw    $14, STK_OFFSET_GPR14($29)

lw    $13, STK_OFFSET_GPR13($29)

lw    $12, STK_OFFSET_GPR12($29)

lw    $11, STK_OFFSET_GPR11($29)

lw    $10, STK_OFFSET_GPR10($29)

lw    $9,  STK_OFFSET_GPR9($29)

lw    $8,  STK_OFFSET_GPR8($29)

lw    $7,  STK_OFFSET_GPR7($29)

lw    $6,  STK_OFFSET_GPR6($29)

lw    $5,  STK_OFFSET_GPR5($29)

lw    $4,  STK_OFFSET_GPR4($29)
```

```

lw      $3,  STK_OFFSET_GPR3($29)
lw      $2,  STK_OFFSET_GPR2($29)
lw      $1,  STK_OFFSET_GPR1($29)

addi  $29, $29, STK_CTX_SIZE      /* 调整堆栈指针 */

*****
*** 第四段：返回
***** */
****

eret          /* 返回 */

.end OSIntCtxSw

```

上述代码可以分为四段理解。

第一段：调用钩子函数OSTaskSwHook，该函数在文件os_cpu_c.c中定义，默认为空，用户可以在其中填充自己的代码，每次任务切换都会调用该函数。

第二段：更新当前任务的优先级OSPrioCur及其任务控制块指针OSTCBCur这两个变量，分别等于目前处于就绪态的具有最高优先级的任务的优先级OSPrioHighRdy及其任务控制块指针OSTCBHighRdy。

第三段：恢复新任务的寄存器。从图15-12可知，就绪态任务的堆栈指针就是任务控制块的第一个元素，所以可以从任务控制块中获取堆栈指针。之后，从堆栈中恢复各个寄存器。

第四段：使用指令eret返回，指令eret会将寄存器EPC的值赋给取指寄存器PC，作为新的取指地址。

如此就实现了任务切换。

7. 函数ExceptionHandler

在前文定义的系统调用异常、无效指令、溢出异常、自陷异常的处理例程中，会直接转移到函数ExceptionHandler，进行异常处理。函数ExceptionHandler的定义如下。

```
.ent ExceptionHandler
ExceptionHandler:

/*****
**
*****          第一段：保护现场，寄存器压栈          *****
*/
addi  $29, $29, -STK_CTX_SIZE      /* 调整堆栈指针 */

sw    $1, STK_OFFSET_GPR1($29)      /* 保存整数寄存器 */
sw    $2, STK_OFFSET_GPR2($29)
```

```
SW      $3,  STK_OFFSET_GPR3($29)
SW      $4,  STK_OFFSET_GPR4($29)
SW      $5,  STK_OFFSET_GPR5($29)
SW      $6,  STK_OFFSET_GPR6($29)
SW      $7,  STK_OFFSET_GPR7($29)
SW      $8,  STK_OFFSET_GPR8($29)
SW      $9,  STK_OFFSET_GPR9($29)
SW      $10, STK_OFFSET_GPR10($29)
SW      $11, STK_OFFSET_GPR11($29)
SW      $12, STK_OFFSET_GPR12($29)
SW      $13, STK_OFFSET_GPR13($29)
SW      $14, STK_OFFSET_GPR14($29)
SW      $15, STK_OFFSET_GPR15($29)
SW      $16, STK_OFFSET_GPR16($29)
SW      $17, STK_OFFSET_GPR17($29)
SW      $18, STK_OFFSET_GPR18($29)
SW      $19, STK_OFFSET_GPR19($29)
SW      $20, STK_OFFSET_GPR20($29)
SW      $21, STK_OFFSET_GPR21($29)
SW      $22, STK_OFFSET_GPR22($29)
SW      $23, STK_OFFSET_GPR23($29)
SW      $24, STK_OFFSET_GPR24($29)
SW      $25, STK_OFFSET_GPR25($29)
SW      $26, STK_OFFSET_GPR26($29)
SW      $27, STK_OFFSET_GPR27($29)
SW      $28, STK_OFFSET_GPR28($29)
SW      $30, STK_OFFSET_GPR30($29)
```

```
        sw      $31,  STK_OFFSET_GPR31($29)

        mflo   $8
        mfhi   $9
        sw      $8,   STK_OFFSET_LO($29)          /* 保存寄存器LO */
        sw      $9,   STK_OFFSET_HI($29)          /* 保存寄存器HI */

        mfc0   $8,   $14,  0
        addi   $8,   $8,   4
        sw      $8,   STK_OFFSET_EPC($29)         /* 保存寄存器EPC */

        mfc0   $8,   $12,  0
        sw      $8,   STK_OFFSET_SR($29)          /* 保存寄存器SR */

        la     $10,  OSTCBCur
        lw     $11,  0($10)
        sw      $29,  0($11)                      /* 保存堆栈指针到任务控制块
*/
/* ****
** *
***** 第二段：异常处理 *****

**** */
        la     $8,   BSP_Exception_Handler
```

```
jalr    $8          /* 调用函数BSP_Exception_Handler进行  
异常处理 */  
  
nop  
  
/*  
**  
***** 第三段：恢复现场 *****  
*****  
*/  
  
la      $10,  OSTCBCur  
lw      $9,   0($10)  
lw      $29,  0($9)          /* 恢复新任务的堆栈指针到寄  
存器sp */  
  
lw      $8,   STK_OFFSET_SR($29)  
mtc0  $8,   $12,  0          /* 恢复寄存器SR */  
  
lw      $8,   STK_OFFSET_EPC($29)  
mtc0  $8,   $14,  0          /* 恢复寄存器EPC */  
  
lw      $8,   STK_OFFSET_L0($29)  
lw      $9,   STK_OFFSET_HI($29)  
mtlo  $8          /* 恢复寄存器LO */  
mtlo  $9          /* 恢复寄存器HI */  
  
lw      $31,  STK_OFFSET_GPR31($29)  /* 恢复整数寄存器 */
```

lw	\$30, STK_OFFSET_GPR30(\$29)
lw	\$28, STK_OFFSET_GPR28(\$29)
lw	\$27, STK_OFFSET_GPR27(\$29)
lw	\$26, STK_OFFSET_GPR26(\$29)
lw	\$25, STK_OFFSET_GPR25(\$29)
lw	\$24, STK_OFFSET_GPR24(\$29)
lw	\$23, STK_OFFSET_GPR23(\$29)
lw	\$22, STK_OFFSET_GPR22(\$29)
lw	\$21, STK_OFFSET_GPR21(\$29)
lw	\$20, STK_OFFSET_GPR20(\$29)
lw	\$19, STK_OFFSET_GPR19(\$29)
lw	\$18, STK_OFFSET_GPR18(\$29)
lw	\$17, STK_OFFSET_GPR17(\$29)
lw	\$16, STK_OFFSET_GPR16(\$29)
lw	\$15, STK_OFFSET_GPR15(\$29)
lw	\$14, STK_OFFSET_GPR14(\$29)
lw	\$13, STK_OFFSET_GPR13(\$29)
lw	\$12, STK_OFFSET_GPR12(\$29)
lw	\$11, STK_OFFSET_GPR11(\$29)
lw	\$10, STK_OFFSET_GPR10(\$29)
lw	\$9, STK_OFFSET_GPR9(\$29)
lw	\$8, STK_OFFSET_GPR8(\$29)
lw	\$7, STK_OFFSET_GPR7(\$29)
lw	\$6, STK_OFFSET_GPR6(\$29)
lw	\$5, STK_OFFSET_GPR5(\$29)
lw	\$4, STK_OFFSET_GPR4(\$29)
lw	\$3, STK_OFFSET_GPR3(\$29)

```
lw      $2,    STK_OFFSET_GPR2($29)
lw      $1,    STK_OFFSET_GPR1($29)

addi  $29, $29, STK_CTX_SIZE           /* 调整堆栈指针 */

/********************* 第四段：返回 ********************

eret          /* 返回 */
.end ExceptionHandler
```

函数ExceptionHandler与函数InterruptHandler的内容大致一样，但是更加简单，可分为四段理解。

第一段：保护现场，就是将各个寄存器保存到堆栈，同时将当前任务的堆栈指针sp（即寄存器\$29）保存到任务控制块OSTCBCur中，保存后的格式就是图15-12所示。

第二段：调用具体的异常处理函数BSP_Exception_Handler进行异常处理。该函数在文件os_cpu_c.c中定义，在15.11.4节会有详细说明。其中将进行任务切换。

第三段：由于在函数BSP_Exception_Handler中会进行任务切换，所以从函数BSP_Exception_Handler返回后，OSTCBCur可能不是指向被异常打断的任务的任务控制块。不管是不是，都要从OSTCBCur中获得将要执行的任务的堆栈指针，然后依据该堆栈指针恢复各个寄存器。

第四段：使用指令eret返回，eret会将寄存器EPC的值赋给取指寄存器PC，作为新的取指地址。

8. 函数OSStartHighRdy

μ C/OS-II在启动前要至少创建一个任务，新创建的任务处于就绪态。然后 μ C/OS-II可以调用函数OSStart实现启动。OSStart从任务就绪表中找出用户建立的优先级最高的任务，并调用函数OSStartHighRdy以执行该优先级最高的任务。OSStartHighRdy的代码如下。

```
.ent OSStartHighRdy
OSStartHighRdy:
/*****
**
*****      第一段：调用钩子函数OSTaskSwHook      *****
*/
la    $8,  OSTaskSwHook
jalr $8          /* 调用钩子函数OSTaskSwHook */
nop
```

```
/*********************  
**  
***** 第二段：设置操作系统运行标志OSRunning *****  
*****  
**/  
  
    addi $8, $0, 1  
    la    $9, OSRunning  
    sb    $8, 0($9)          /* 设置OSRunning为1，表示操作系  
统在运行中 */  
  
/*********************  
**  
***** 第三段：堆栈恢复 *****  
*****  
**/  
  
    la    $8, OSTCBHighRdy  
    lw    $9, 0($8)          /* 得到当前就绪的最高优先级任务的  
任务控制块 */  
    lw    $29, 0($9)          /* 得到当前就绪的最高优先级任务的  
堆栈指针 */  
  
    lw    $8, STK_OFFSET_SR($29)  
    mtc0 $8, $12, 0          /* 恢复寄存器SR */
```

```
lw      $8,  STK_OFFSET_EPC($29)
mtc0   $8,  $14, 0          /* 恢复寄存器EPC */

lw      $8,  STK_OFFSET_L0($29)
lw      $9,  STK_OFFSET_HI($29)
mtlo   $8          /* 恢复寄存器LO */
mthi   $9          /* 恢复寄存器HI */

lw      $31, STK_OFFSET_GPR31($29) /* 恢复整数寄存器 */
lw      $30, STK_OFFSET_GPR30($29)
lw      $28, STK_OFFSET_GPR28($29)
lw      $27, STK_OFFSET_GPR27($29)
lw      $26, STK_OFFSET_GPR26($29)
lw      $25, STK_OFFSET_GPR25($29)
lw      $24, STK_OFFSET_GPR24($29)
lw      $23, STK_OFFSET_GPR23($29)
lw      $22, STK_OFFSET_GPR22($29)
lw      $21, STK_OFFSET_GPR21($29)
lw      $20, STK_OFFSET_GPR20($29)
lw      $19, STK_OFFSET_GPR19($29)
lw      $18, STK_OFFSET_GPR18($29)
lw      $17, STK_OFFSET_GPR17($29)
lw      $16, STK_OFFSET_GPR16($29)
lw      $15, STK_OFFSET_GPR15($29)
lw      $14, STK_OFFSET_GPR14($29)
lw      $13, STK_OFFSET_GPR13($29)
lw      $12, STK_OFFSET_GPR12($29)
```

```

lw      $11, STK_OFFSET_GPR11($29)
lw      $10, STK_OFFSET_GPR10($29)
lw      $9,   STK_OFFSET_GPR9($29)
lw      $8,   STK_OFFSET_GPR8($29)
lw      $7,   STK_OFFSET_GPR7($29)
lw      $6,   STK_OFFSET_GPR6($29)
lw      $5,   STK_OFFSET_GPR5($29)
lw      $4,   STK_OFFSET_GPR4($29)
lw      $3,   STK_OFFSET_GPR3($29)
lw      $2,   STK_OFFSET_GPR2($29)
lw      $1,   STK_OFFSET_GPR1($29)

/ ****
**                                         第四段：跳转到任务
*****
****

jr      $31          /* 进入优先级最高的任务执行
*/
addi   $29, $29, STK_CTX_SIZE      /* 调整堆栈指针 */

.end OSStartHighRdy

```

从代码上分析，函数OSStartHighRdy的主要工作就是从当前最高优先级的任务的堆栈恢复各个寄存器。这一过程与之前介绍的函数

OSIntCtxSw是一样的，两者的代码也极为相似。可以分为四段理解。

第一段：调用钩子函数OSTaskSwHook，该函数在文件os_cpu_c.c中定义，其内容默认为空，用户可以填充自己的代码。

第二段：设置操作系统运行标志OSRunning为True（即1），表示操作系统开始运行了。

第三段：恢复处于就绪态的具有最高优先级的任务的寄存器。从图15-12可知，就绪态任务的堆栈指针放在任务控制块中，所以首先从任务控制块中获取堆栈指针，然后可以从堆栈中恢复各个寄存器。

第四段：使用jr指令返回。这一点是与函数OSIntCtxSw的最大不同，后者使用eret指令返回，进入新任务执行。而此处是使用jr指令进入新任务执行。这是因为在新任务创建的时候，会将任务的入口地址存放在寄存器\$31中，所以此处使用jr \$31指令，就会转移到新任务开始执行。读者在阅读15.11.4节解释函数OSTaskStkInit的时候，能够对此有更直观的体会。

9. 函数TickInterruptClear

当协处理器CP0中的Compare寄存器等于Count寄存器时，会引发时钟中断，时钟中断会持续声明，直到修改了Compare寄存器的值。函数TickInterruptClear的作用就是通过将Compare寄存器置为0，从而清除时钟中断声明。其代码如下。

```
.ent TickInterruptClear
TickInterruptClear:
```

```
    mtc0 $0, $11      /* 设置Compare寄存器为0 */
    jr $31           /* 返回 */
    nop

.end TickInterruptClear
```

10. 函数CoreTmrInit

本函数负责初始化定时器，也就是设置Compare寄存器的值，同时将Count寄存器清零。函数声明如下，从中可知该函数有一个输入参数tmr_reload，就是要设置给Compare寄存器的值。

```
void CoreTmrInit(CPU_INT32U tmr_reload);
```

函数代码如下。其中将寄存器\$4的值赋给寄存器Compare，参考15.10.2节MIPS函数调用中的参数传递规范可知，寄存器\$4中保存的就是传入的参数tmr_reload。

```
.ent CoreTmrInit
CoreTmrInit:

    mtc0 $4, $11      /* 设置Compare寄存器 */
    nop
    mtc0 $0, $9       /* 将Count寄存器清零 */
    jr $31
    nop

.end CoreTmrInit
```

11. 函数TickISR

中断发生时会调用函数BSP Interrupt Handler进行处理，如果是时钟中断，那么会进一步调用本函数TickISR进行处理，其作用是增加Compare寄存器的值，这样不仅清除了时钟中断声明，而且新加的数据还确定了下一次时钟中断的发生时刻（当寄存器Count等于寄存器Compare的新值时，又会发生时钟中断）。函数声明如下，从中可知该函数有一个输入参数tmr_reload，就是要给寄存器Compare增加的值。

```
void TickISR(CPU_INT32U tmr_reload);
```

函数代码如下。

```
.ent TickISR
TickISR:

/ ****
**
***** 第一段：保存部分寄存器 *****
**/


addiu $29,$29,-24
sw    $16, 0x4($29)
sw    $8,   0x8($29)
sw    $31,  0xC($29) /* 将寄存器$8、$16、$31压栈 */
```

```
*****  
**  
***** 第二段：修改寄存器Compre *****  
*****  
**/  
  
    mfco $8, $11      /* 获得当前Compare寄存器的值 */  
    addu $8, $4       /* 加上传入的参数 */  
    mtc0 $8, $11      /* 加法的结果作为Compare寄存器的新值 */  
  
*****  
**  
***** 第三段：调用函数OSTimeTick *****  
*****  
**/  
  
    la $8, OSTimeTick  
    jalr $8           /* 调用函数OSTimeTick以通知操作系统有一个  
时钟中断 */  
    nop  
  
*****  
**  
***** 第四段：恢复部分寄存器 *****  
*****  
**/
```

```

lw $31, 0xC($29)           /* 从堆栈恢复寄存器$8、$16、$31 */
lw $16, 0x4($29)
lw $8, 0x8($29)
addiu $29,$29,24

/ ****
**          第五段：返回
*****
****

jr      $31           /* 返回 */
nop

.end TickISR

```

上述代码可以分为五段理解。

第一段：因为在函数中又要调用函数OSTimeTick，所以需要保存部分寄存器。首先将堆栈指针sp向下移，预留24个字节的空间。然后，把函数中使用到的寄存器\$8、\$16、\$31保存到堆栈。

第二段：获取Compare寄存器的当前值，然后与寄存器\$4相加，加法的结果再保存回Compare寄存器。参考15.10.2节MIPS函数调用中的参数传递规范可知，寄存器\$4中保存的就是传入参数tmr_reload。

第三段：调用函数OSTimeTick，以通知操作系统有一个时钟中断发生，该函数不需修改。

第四段：从堆栈恢复保存的寄存器。

第五段：返回。

截止到目前，已经提到了好几个与中断有关的处理函数，包括InterruptHandler、BSP Interrupt Handler、OSIntExit、OSIntCtxSw、TickISR等，其中函数OSIntExit位于μC/OS-II中与处理器无关的部分，无需修改，函数BSP Interrupt Handler将在15.11.4节介绍，其余的3个函数都已介绍，读者可能觉得它们之间的关系比较混乱，在此做一说明。这几个函数的关系如图15-13所示。

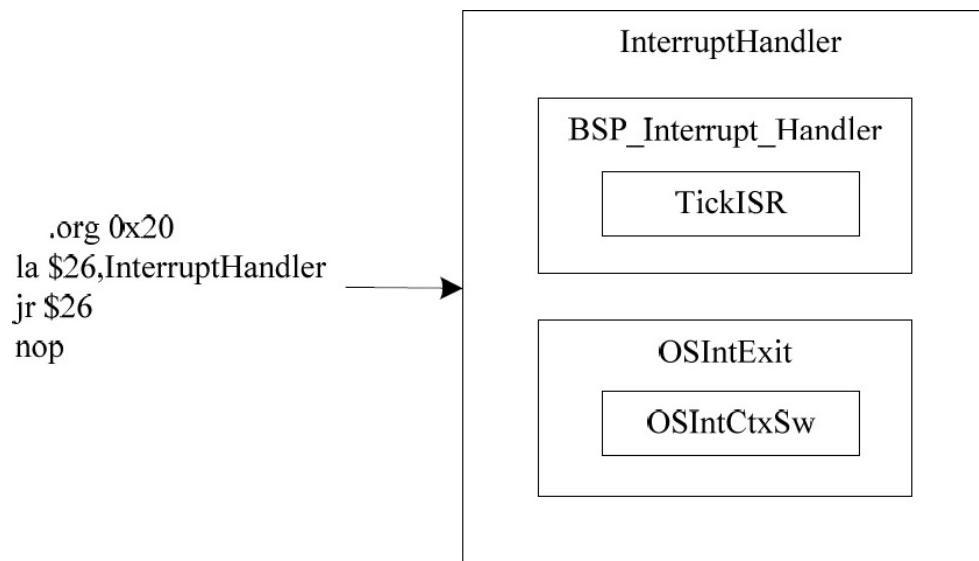


图15-13 中断处理相关函数的关系

中断发生时，首先转移到0x20处，然后跳转到函数InterruptHandler，在其中调用函数BSP Interrupt Handler按照中断种类进行具体处理，如果是时钟中断，那么还要调用函数TickISR以清除时钟中断声明，同时设置下一次时钟中断的发生时刻。从函数

BSP Interrupt Handler 返回到 InterruptHandler 后，会再调用函数 OSIntExit，如果全局变量 OSIntNesting 为 0，且有更高优先级的任务处于就绪态，那么还会调用函数 OSIntCtxSw 进行任务切换，反之，返回到函数 InterruptHandler，恢复被中断的任务继续执行。上述就是中断处理过程涉及到的主要函数的关系。

12. 函数 DisableInterruptSource

本函数的作用是通过设置 Status 寄存器中 IM 字段指定位为 0，从而禁止指定的外部中断。函数声明如下。

```
void DisableInterruptSource(CPU_INT32U int_source);
```

函数代码如下。其中将寄存器 \$4 的值与 Compare 寄存器相与，结果再保存回 Compare 寄存器，参考 15.10.2 节 MIPS 函数调用的参数传递规范可知，寄存器 \$4 中保存的就是传入的参数 int_source。

```
.ent DisableInterruptSource
DisableInterruptSource:

    mfc0  $8,    $12      /* 获取 Status 寄存器的值 */
    and    $8,    $4       /* 与传入的参数相与 */
    mtc0  $8,    $12      /* 结果再保存回 Status 寄存器 */
    jr    $31
    nop

.end DisableInterruptSource
```

13. 函数 EnableInterruptSource

本函数的作用是通过设置Status寄存器中IM字段指定位为1，从而使能指定的外部中断。函数声明如下。

```
void EnableInterruptSource(CPU_INT32U int_source);
```

函数代码如下。其中将寄存器\$4的值与Compare寄存器相或，结果再保存回Compare寄存器，参考15.10.2节MIPS函数调用的参数传递规范可知，寄存器\$4中保存的就是传入的参数int_source。

```
.ent DisableInterruptSource
DisableInterruptSource:

    mfc0  $8,    $12      /* 获取Status寄存器的值 */
    or     $8,    $4       /* 与传入的参数相或 */
    mtc0  $8,    $12      /* 结果再保存回Status寄存器 */
    jr     $31
    nop

.end DisableInterruptSource
```

本节给出的代码位于本书附带光盘Code\Chapter15\ucosii_OpenMIPS\port目录下的同名文件中。

15.11.4 修改os_cpu_c.c文件

移植过程最后一个要修改的文件是os_cpu_c.c，从后缀就可以知道这是一个C语言文件。包括如下几个函数。

- OSInitHookBegin
- OSInitHookEnd
- OSTaskCreateHook
- OSTaskDelHook
- OSTaskReturnHook
- OSTaskIdleHook
- OSTaskStatHook
- OSTaskSwHook
- OSTCBInitHook
- OSTimeTickHook
- OSTaskStkInit
- BSP_Interrupt_Handler
- BSP_Exception_Handler

其中，函数名有“Hook”单词的表示该函数是一个钩子函数，这类函数的内容可以保持为空，移植过程中不需要修改，本节不再提及。只解释最后三个函数。

1. 函数OSTaskStkInit

函数OSTaskStkInit被函数OSTaskCreate或者函数OSTaskCreateExt调用，用来初始化即将创建的任务的堆栈。函数声明如下。

```
OS_STK *OSTaskStkInit (void (*task)(void *pd),  
                      void *p_arg,  
                      OS_STK *ptos,  
                      INT16U opt);
```

从中可以发现有四个参数，分别如下。

- (1) task：指向任务的执行代码。
- (2) p_arg：指向用户提供的一段内存区域。
- (3) ptos：指向堆栈顶部。
- (4) opt：一些特殊选项。

函数OSTaskStkInit的返回值是堆栈栈顶地址。本函数用来初始化即将创建的任务的堆栈，而μC/OS-II中新创建的任务处于就绪态，其堆栈结构应该如图15-10所示。有了这个知识，就容易理解函数OSTaskStkInit了，其代码如下。

```
OS_STK *OSTaskStkInit (void (*task)(void *pd),
                        void      *p_arg,
                        OS_STK   *ptos,
                        INT16U   opt)

{
    INT32U    *pstk;
    INT32U    sr_val;
    INT32U    gp_val;

    /* 函数中并没有使用参数opt，下面代码的作用是防止编译器给出告警信息 */
    (void)opt;

    /* 获取Status寄存器的值，此处使用到了C语言内嵌汇编，其语法结构如15.9
节所述*/
}
```

```
asm volatile("mfc0    %0,$12"    : "=r"(sr_val));
```

/* Status寄存器的值保存在变量sr_val中，设置其第10位为1，设置其第0位
也

为1，sr_val将作为新任务的对应Status寄存器的值，此处的设置就是使得
新任

务在执行时允许时钟中断 */

```
sr_val |= 0x00000401;
```

```
/* 获取当前全局指针寄存器gp的值，也就是寄存器%28 */
```

```
asm volatile("addi    %0,$28,0" : "=r"(gp_val));
```

```
/* 设置变量pstk为传入的参数ptos的值，即堆栈的栈顶地址 */
```

```
pstk = (INT32U *)ptos;
```

/* μC/OS-II的堆栈是从高地址向低地址生长的，下面的代码中，pstk依次降
低，设置堆栈

的初始值，这些值对应图15-10中的各个寄存器，其中大部分寄存器的初始
值都可以任意

设置 */

```
pstk--;
```

```
*pstk-- = (INT32U)task;           /* 整数寄存器$31 */
```

```
*pstk-- = (INT32U)0x30303030;    /* 整数寄存器$30 */
```

```
*pstk-- = gp_val;                /* 整数寄存器$28 */
```

```
*pstk-- = (INT32U)0x27272727;    /* 整数寄存器$27 */
```

```
*pstk-- = (INT32U)0x26262626;    /* 整数寄存器$26 */
```

```
*pstk-- = (INT32U)0x25252525;    /* 整数寄存器$25 */
```

```
*pstk-- = (INT32U)0x24242424; /* 整数寄存器$24 */
*pstk-- = (INT32U)0x23232323; /* 整数寄存器$23 */
*pstk-- = (INT32U)0x22222222; /* 整数寄存器$22 */
*pstk-- = (INT32U)0x21212121; /* 整数寄存器$21 */
*pstk-- = (INT32U)0x20202020; /* 整数寄存器$20 */
*pstk-- = (INT32U)0x19191919; /* 整数寄存器$19 */
*pstk-- = (INT32U)0x18181818; /* 整数寄存器$18 */
*pstk-- = (INT32U)0x17171717; /* 整数寄存器$17 */
*pstk-- = (INT32U)0x16161616; /* 整数寄存器$16 */
*pstk-- = (INT32U)0x15151515; /* 整数寄存器$15 */
*pstk-- = (INT32U)0x14141414; /* 整数寄存器$14 */
*pstk-- = (INT32U)0x13131313; /* 整数寄存器$13 */
*pstk-- = (INT32U)0x12121212; /* 整数寄存器$12 */
*pstk-- = (INT32U)0x11111111; /* 整数寄存器$11 */
*pstk-- = (INT32U)0x10101010; /* 整数寄存器$10 */
*pstk-- = (INT32U)0x09090909; /* 整数寄存器$9 */
*pstk-- = (INT32U)0x08080808; /* 整数寄存器$8 */
*pstk-- = (INT32U)0x07070707; /* 整数寄存器$7 */
*pstk-- = (INT32U)0x06060606; /* 整数寄存器$6 */
*pstk-- = (INT32U)0x05050505; /* 整数寄存器$5 */
*pstk-- = (INT32U)0x04040404; /* 整数寄存器$4 */
*pstk-- = (INT32U)0x03030303; /* 整数寄存器$3 */
*pstk-- = (INT32U)0x02020202; /* 整数寄存器$2 */
*pstk-- = (INT32U)0x01010101; /* 整数寄存器$1 */
*pstk-- = (INT32U)0x00000000; /* 寄存器HI */
*pstk-- = (INT32U)0x00000000; /* 寄存器LO */
*pstk-- = (INT32U)task; /* 寄存器EPC */
```

```
*pstk-- = sr_val; /* 寄存器SR */

return ((OS_STK *)pstk); /* 返回值就堆栈的栈顶地址 */
}
```

详细注释在程序中已经给出，不再重复说明，只提醒读者注意一点，此处将新任务的入口地址task存放到堆栈中对应整数寄存器\$31的位置，所以上一节解释的函数OSStartHighRdy可以在最后使用指令jr \$31跳转到新创建的最高优先级的任务开始执行。

2. 函数BSP_Interrupt_Handler

如图15-13所示，中断发生后，会调用本函数进行具体的中断处理。其代码如下。

```
void BSP_Interrupt_Handler (void)
{
    INT32U cause_val;
    INT32U cause_reg;
    INT32U cause_ip;

    /* 读取Cause寄存器，获得其中的IP (Interrupt Pending) 字段 */
    asm ("mfco %0,$13" : "=r"(cause_val));
    cause_reg = cause_val;
    cause_ip = cause_reg & 0x0000FF00;

    if((cause_ip & 0x00000400) != 0 )
    {
```

```
/* 如果IP字段表示是时钟中断，那么调用函数TickISR，在该函数中将
   Compare寄存器增加0x50000，同时清除时钟中断声明 */
TickISR(0x50000);

}

}
```

上述代码首先获取Cause寄存器的IP字段，据此可以了解中断类型，然后分别进行处理。此处只对时钟中断进行处理，读者朋友可以自行添加对其余中断的处理代码。如果是时钟中断，那么会调用函数TickISR，该函数在文件os_cpu_a.S中已经解释，作用是修改Compare寄存器的值，并清除时钟中断声明。此处将Compare寄存器的值增加0x50000。这样下一次时钟中断就会在0x50000个时钟周期后再次发生，如果系统的运行频率是27MHz，那么0x50000个时钟周期大概是12ms。

3. 函数BSP_Exception_Handler

系统调用异常、无效指令、溢出异常、自陷异常等异常发生后，会进入函数ExceptionHandler进行处理，后者会调用本函数处理各种具体异常。代码如下。

```
void  BSP_Exception_Handler (void)
{
    INT32U    cause_val;
    INT32U    cause_exccode;
    INT32U    EPC;

    /* 读取Cause寄存器，获得其中的ExcCode字段，该字段存储的是异常原因
```

```
*/  
  
asm volatile("mfcc0 %0, $13" : "=r"(cause_val));  
  
cause_exccode = (cause_val & 0x0000007C);  
  
if(cause_exccode == 0x00000020) /* 判断是否是由syscall  
指令引起 */  
{  
    OSIntCtxSw();  
}  
else if(cause_exccode == 0x00000034) /* 判断是否是由于自陷指令  
引起 */  
{  
    OSIntCtxSw();  
}  
else if(cause_exccode == 0x00000030) /* 判断是否是由于溢出引起  
*/  
{  
    OSIntCtxSw();  
}  
else if(cause_exccode == 0x00000028) /* 判断是否是由于无效指令  
引起 */  
{  
    OSIntCtxSw();  
}  
}
```

现在，读者可以回顾在os_cpu.h中定义的一个宏，如下。

```
#define OS_TASK_SW()      asm("\tsyscall\n");
```

如上所示的宏用来进行任务切换，是在μC/OS-II从低优先级任务切换到高优先级任务时使用到的。它是如何实现任务切换的呢？在这里可以找到答案。

从定义中可以发现该宏定义实际就是系统调用指令syscall，该指令会引起系统调用异常。从函数BSP_Exception_Handler的代码可知，系统调用异常的处理过程会调用函数OSIntCtxSw。函数OSIntCtxSw在文件os_cpu_a.S中实现，上一小节已给出解释说明，其作用就是进行任务切换。由此，实现了宏OS_TASK_SW()的功能。

对于无效指令、溢出异常、自陷异常等异常情况，并没有明确如何处理，读者可以依据自己的需要灵活修改，在笔者的移植过程中，只进行任务切换。

本节给出的代码位于本书附带光盘中Code\Chapter15\ucosii_OpenMIPS\port目录下的同名文件。

上述通过对os_cpu.h、os_cpu_a.S、os_cpu_c.c等三个文件的修改，实现了移植μC/OS-II到OpenMIPS处理器。下面将编写测试程序以验证μC/OS-II是否移植成功。

15.12 测试程序

本节将编写一个简单的测试程序，在该程序中，我们创建了一个用户任务，该任务通过UART输出一个指定字符串中的一个字符，同时通过GPIO输出一个数（初始的时候为0），然后延时大概100ms，再

输出指定字符串中的下一个字符，同时将GPIO的输出加2。为实现该测试，需要创建两个文件openmips.h、openmips.c。同时，修改include文件夹下的文件includes.h，在其中引用新创建的文件openmips.h，如下。

```
#include <stdarg.h>
#include <stddef.h>
#include <limits.h>
#include "ucos_ii.h"

#include "openmips.h"      //增加对openmips.h文件的引用
```

15.12.1 创建openmips.h文件

openmips.h文件定义了一些在测试程序中会使用到的宏定义，主要内容如下。源文件位于本书附带光盘中Code\Chapter15\ucosii_OpenMIPS\include目录下。

```
*****
**
*****
***** 第一段：三种加载、存储
*****
*****
```

```
**/  
  
#define REG8(addr)  *((volatile INT8U *) (addr))  
#define REG16(addr) *((volatile INT16U *) (addr))  
#define REG32(addr) *((volatile INT32U *) (addr))  
  
/******  
**  
*****  
* 第二段：系统时钟 *  
*****  
**/  
  
#define IN_CLK 27000000 /* 输入时钟是27MHz */  
  
/******  
**  
*****  
* 第三段：与UART控制器有关的宏 *  
*****  
**/  
  
#define UART_BAUD_RATE 9600 /* UART速率是9600bps */  
#define UART_BASE 0x10000000 /* UART控制器的起始地址 */  
#define UART_LC_REG 0x00000003 /* Line Control寄存器的偏移  
地址 */  
#define UART_IE_REG 0x00000001 /* Interrupt Enable寄存器  
的偏移地址 */  
#define UART_TH_REG 0x00000000 /* Transmitter Holding寄存
```

```

器的偏移地址*/
#define UART_LS_REG      0x00000005 /* Line Status寄存器的偏移
地址 */
#define UART_DLBI_REG    0x00000000 /* 分频系数低字节的偏移地址
*/
#define UART_DLBB_REG    0x00000001 /* 分频系数高字节的偏移地址
*/

/* Line Status寄存器的标志位 */
#define UART_LS_TEMT 0x40 /* 第6bit为发送数据空标志 */
#define UART_LS_THRE 0x20 /* 第5bit为发送FIFO空标志 */

/* Line Control寄存器的标志位 */
#define UART_LC_NO_PARITY 0x00 /* 第3bit为0，表示禁止奇偶校验 */
#define UART_LC_ONE_STOP 0x00 /* 第2bit为0，表示1位停止位 */
#define UART_LC_WLEN8     0x03 /* 最低两位为11，表示数据长度是8位
*/
/* 一些函数声明 */
extern void uart_init(void); /* UART控制器初始化函数 */
extern void uart_putc(char); /* UART控制器输出字节函数 */
extern void uart_print_str(char*); /* UART控制器输出字符串函数 */

//*****
** ****
***** 第四段：与GPIO模块有关的宏 ****
*****
```

```

/**/

#define GPIO_BASE      0x20000000 /* GPIO模块的起始地址 */
#define GPIO_IN_REG    0x00000000 /* GPIO模块输入寄存器的偏移地址
*/
#define GPIO_OUT_REG   0x00000004 /* GPIO模块输出寄存器的偏移地址
*/
#define GPIO_OE_REG    0x00000008 /* GPIO模块输出使能寄存器的偏移
地址 */
#define GPIO_INTE_REG  0x0000000c /* GPIO模块中断使能寄存器的偏移
地址 */

/* 一些函数声明 */
extern void gpio_init(void);          /* GPIO模块初始化函数 */
extern void gpio_out(INT32U);         /* GPIO模块输出函数 */
extern INT32U gpio_in(void);          /* 读取GPIO模块输入的函数 */

***** **
***** 第五段：主函数main声明 *****
***** /


extern void main(void);

```

上述代码可以分为五段理解。

第一段：定义了三种加载、存储的宏，分别是：字节、半字、字。通过如下实例说明其用法。

```
result      = REG8(addr);      //加载地址addr处的字节  
result      = REG16(addr);     //加载地址addr处的半字  
result      = REG32(addr);     //加载地址addr处的字  
REG8(addr)  = 0xFF;           //将字节0xFF存储到地址addr处  
REG16(addr) = 0xFFFF;         //将半字0xFFFF存储到地址addr处  
REG32(addr) = 0xFFFFFFFF;    //将字0xFFFFFFFF存储到地址addr处
```

第二段：定义了系统时钟，我们的最小SOPC运行在DE2平台上，使用的是27MHz的时钟，所以此处设置为27000000。

第三段：定义了与UART控制器有关的宏，包括：UART控制器的起始地址，因为在小型SOPC中，UART控制器挂在WB_CONMAX的从设备接口1，所以其地址空间从0x10000000开始；还定义了UART控制器中各个寄存器的地址相对起始地址的偏移；还定义了UART控制器的传输速率，此处设置为9600bps。另外，还有一些标志位，读者可以参考13.4.2节理解，在文件openmips.c使用这些标志位的时候会更加清楚其含义。

第四段：定义了与GPIO模块有关的宏，包括：GPIO模块的起始地址，因为在小型SOPC中，GPIO模块挂在WB_CONMAX的从设备接口2，所以其地址空间从0x20000000开始；还定义了GPIO模块中各个寄存器的地址相对起始地址的偏移。

第五段：主函数main声明。

15.12.2 创建openmips.c文件

openmips.c文件实现了用户任务，该任务通过UART输出一个指定字符串中的一个字符，同时通过GPIO输出一个数（初始的时候为0），然后延时大概100ms，再输出指定字符串中的下一个字符，同时将GPIO的输出加2。主要代码如下，源文件位于本书附带光盘中Code\Chapter15\ucosii_OpenMIPS\common目录下。

```
*****  
**  
***** 第一段：一些宏定义  
*****  
*****  
**/  
  
#include "includes.h"  
  
#define BOTH_EMPTY (UART_LS_TEMT | UART_LS_THRE)  
  
/* 循环等待，直到UART控制器的发送FIFO为空、移位寄存器为空，表示数据发送完  
毕 */  
#define  
WAIT_FOR_XMITR \  
*****
```

```
do { \
    lsr = REG8(UART_BASE + UART_LS_REG); \
} while ((lsr & BOTH_EMPTY) != BOTH_EMPTY)

/* 循环等待，直到UART控制器发送FIFO为空，此时不一定发送完毕，但是可以接着
通过 UART控制器发送数据 */
#define WAIT_FOR_THRE

\

do { \
    lsr = REG8(UART_BASE + UART_LS_REG); \
} while ((lsr & UART_LS_THRE) != UART_LS_THRE)

/* 给用户任务使用的堆栈，大小是256个字，其中OS_STK就是int类型，其在
os_cpu.h
中定义，参考15.11.2节 */
#define TASK_STK_SIZE 256

OS_STK
TaskStartStk

[TASK_STK_SIZE];
```

```
/* 要通过UART发送的字符串 */
char
Info[103]

={0xC9, 0xCF, 0xB5, 0xDB, 0xCB, 0xB5, 0xD2, 0xAA, 0xD3, 0xD0, 0xB9,
 0xE2, 0xA3, 0xAC, 0xD3, 0xDA, 0xCA, 0xC7, 0xBE, 0xCD, 0xD3, 0xD0,
 0xC1, 0xCB, 0xB9, 0xE2, 0x0D, 0x0A, 0xC9, 0xCF, 0xB5, 0xDB, 0xCB,
 0xB5, 0xD2, 0xAA, 0xD3, 0xD0, 0xCC, 0xEC, 0xBF, 0xD5, 0xA3, 0xAC,
 0xD3, 0xDA, 0xCA, 0xC7, 0xBE, 0xCD, 0xD3, 0xD0, 0xC1, 0xCB, 0xCC,
 0xEC, 0xBF, 0xD5, 0x0D, 0x0A, 0xC9, 0xCF, 0xB5, 0xDB, 0xCB, 0xB5,
 0xD2, 0xAA, 0xD3, 0xD0, 0xC2, 0xBD, 0xB5, 0xD8, 0xBA, 0xCD, 0xBA,
 0xA3, 0xD1, 0xF3, 0xA3, 0xAC, 0xD3, 0xDA, 0xCA, 0xC7, 0xBE, 0xCD,
 0xD3, 0xD0, 0xC1, 0xCB, 0xC2, 0xBD, 0xB5, 0xD8, 0xBA, 0xCD, 0xBA,
 0xA3, 0xD1, 0xF3, 0x0D};
```

```
**  
***** 第二段：与UART控制器相关的函数定义 *****  
*****  
**/  
  
void uart_init(void)          /* UART控制器初始化函数 */  
  
{  
    INT32U divisor;  
  
    /* 计算分频系数 */  
    divisor = (INT32U) IN_CLK/(16 * UART_BAUD_RATE);  
  
    /* 设置分频系数寄存器 */  
    REG8(UART_BASE + UART_LC_REG) = 0x80;  
    REG8(UART_BASE + UART_DLBI_REG) = divisor &  
0x000000ff;  
    REG8(UART_BASE + UART_DLBB_REG) = (divisor >> 8) &  
0x000000ff;  
    REG8(UART_BASE + UART_LC_REG) = 0x00;  
  
    /* 禁止UART控制器的所有中断 */  
    REG8(UART_BASE + UART_IE_REG) = 0x00;
```

```
/* 设置数据格式：8位数据位、1位停止位、没有奇偶校验位 */
REG8(UART_BASE + UART_LC_REG) = UART_LC_WLEN8 |
                                (UART_LC_ONE_STOP |
                                 UART_LC_NO_PARITY);

/* 通过UART输出UART控制器初始化完毕信息 */
uart_print_str("UART initialize done ! \n");
return;
}

void uart_putc(char c)          /* 通过UART输出字节 */
{
    unsigned char lsr;
    WAIT_FOR_THRE;           /* 等待发送FIFO空 */
    REG8(UART_BASE + UART_TH_REG) = c;      /* 通过UART输出字节 */
}
if(c == '\n') {                /* 如果是换行符，那么增加一个回车
    WAIT_FOR_THRE;
    REG8(UART_BASE + UART_TH_REG) = '\r';  /* 通过UART输出
```

```
回车符 /*  
}  
WAIT_FOR_XMITR; /* 等待发送数据完毕 */  
  
}  
  
void uart_print_str(char* str) /* 通过UART输出字符串 */  
  
{  
    INT32U i=0;  
    OS_CPU_SR cpu_sr;  
    OS_ENTER_CRITICAL() /*不希望输出字符串的过程被打断，所  
以进入临界区*/  
  
    while(str[i]!=0)  
    {  
        uart_putc(str[i]); /* 调用函数uart_putc依次输出每个  
字节 */  
        i++;  
    }  
  
    OS_EXIT_CRITICAL() /* 输出字符串结束，退出临界区 */
```

```
}

/*****
** 第三段：与GPIO模块相关的函数定义 ****
****/


void gpio_init() /* GPIO模块初始化函数 */

{
    REG32(GPIO_BASE + GPIO_OE_REG) = 0xffffffff; /* 所有输出端口使能 */
    REG32(GPIO_BASE + GPIO_INTE_REG) = 0x00000000; /* 禁用所有中断 */
    gpio_out(0x0f0f0f0f); /* 输出0x0f0f0f0f */

    /* 通过UART输出GPIO模块初始化完毕信息 */
    uart_print_str("GPIO initialize done ! \n");
    return;
}
```

```
void gpio_out(INT32U number)      /* GPIO模块输出函数 */  
  
{  
    REG32(GPIO_BASE + GPIO_OUT_REG) = number;  
}
```

```
INT32U gpio_in()           /* 读取GPIO模块输入的函数 */
```

```
{  
    INT32U temp = 0;  
    temp = REG32(GPIO_BASE + GPIO_IN_REG);  
    return temp;  
}
```

```
*****  
**  
***** 第四段：定时器初始化函数  
*****  
*****  
** /
```

```
void OSInitTick(void)

{

    /* 每个Tick代表一个时钟节拍，会引发一次中断，依据每秒有多少个Tick，计算

        Compare寄存器的初值 */

    INT32U compare = (INT32U)(IN_CLK / OS_TICKS_PER_SEC);

    /* 清零Count寄存器、设置Compare寄存器 */

    asm volatile("mtc0 %0,$9"  : : "r"(0x0));
    asm volatile("mtc0 %0,$11" : : "r"(compare));

    /* 设置Status寄存器，以使能时钟中断 */

    asm volatile("mtc0 %0,$12" : : "r"(0x10000401));

    return;
}
```

```
*****
```

```
**
```

```
*****
```

第五段：用户任务

```
*****
```

```
*****
```

```
/**/  
  
void TaskStart (void *pdata)  
  
{  
    INT32U count = 0;  
    pdata = pdata;  
    OSInitTick();           /* 在用户任务中初始化定时器、允许时钟中断  
 */  
    for (;;) {             /* 一般而言，任务都是一个永不结束的循环 */  
        if(count <= 102)  
        {  
            uart_putc(Info[count]);    /* 输出Info数组中的两个字节，对  
应一个汉字*/  
            uart_putc(Info[count+1]);  
        }  
        gpio_out(count); /* 通过GPIO输出count的值 */  
        count=count+2;    /* count的值加2 */  
        OSTimeDly(10);   /* 等待10个Tick后，再次执行该任务 */  
    }  
}
```

第六段：主函数

```
void main()
{
    OSInit();                                /* μC/OS-II初始化 */
    uart_init();                             /* UART控制器初始化 */
    gpio_init();                            /* GPIO模块初始化 */
    /* 创建用户任务 */
    OSTaskCreate(TaskStart, (void *)0,
                 amp;TaskStartStk[TASK_STK_SIZE - 1], 0);
    OSStart();                               /* μC/OS-II启动 */
}
```

上述实现用户任务的代码比较长，但是结构还是很清晰的，可以分为六段，其中前四段是一些宏、变量的定义，以及与UART控制器、GPIO模块有关的一些函数定义，第五段定义了用户任务，第六段是主函数，在其中创建用户任务、启动μC/OS-II。下面分别介绍。

第一段： 定义了一些宏、变量，如下。

(1) 与UART控制器有关的宏WAIT_FOR_XMITR、WAIT_FOR_THRE。UART控制器在发送数据时，先将要发送数据写入发送FIFO，然后通过移位寄存器依次发送。宏WAIT_FOR_XMITR用来等待数据发送完毕，要求发送FIFO、移位寄存器都为空，才返回；宏WAIT_FOR_THRE仅等待到发送FIFO为空就返回。

以WAIT_FOR_THRE为例解释，在这个宏中，不断读取UART控制器的Line Status寄存器（UART_BASE + UART_LS_REG的和就是Line Status寄存器的地址），判断其发送FIFO空标志是否为1（通过lsr & UART_LS_THRE是否等于UART_LS_THRE判断，其中，UART_LS_THRE的值为0x20，而发送FIFO空标志正是lsr的第5bit）。如果不为1，那么再次读取Line Status寄存器的值，继续判断，直到发送FIFO空标志为1，然后返回。

(2) 定义了任务使用的堆栈TaskStartStk，大小是256个字。

(3) 定义了要通过UART输出的字符串Info，这个字符串是如下文字的十六进制编码，每个汉字使用两个字节表示，所以在后面通过UART发送的时候，需要每次发送两个字节。在Info的最后增加了一个回车符0xD。

第二段：定义了与UART控制器相关的函数。包括UART控制器初始化函数、输出字节函数、输出字符串函数。

(1) UART控制器初始化函数uart_init：该函数与14.6.2节测试程序中的UART控制器初始化过程基本一样，只是后者是使用汇编编写，而此处是使用C语言编写。首先按照UART控制器规定的公式计算分频系数。然后设置Line Control寄存器的最高位为1，此时就可以访问分频系数寄存器了，进而设置两个分频系数寄存器。设置完成后，再将Line Control寄存器的最高位改为0。然后设置数据格式，此处还是8位数据位、1位停止位、没有奇偶校验位。最后，调用函数uart_print_str输出UART控制器初始化完毕信息“UART initialize done!”。

(2) UART输出字节函数uart_putc：该函数首先判断UART的发送FIFO是否为空，如果为空，那么可以发送数据。向UART控制器的Transmitting Holding Register写入要发送的数据，然后调用宏WAIT_FOR_XMITR以等待发送完毕。如果要发送的数据是换行符，那么需要再通过UART发送一个回车符。

(3) UART输出字符串函数uart_print_str：通过调用函数uart_putc依次发送要发送字符串中的每个字节即可。

第三段：定义了与GPIO模块有关的函数。包括GPIO模块初始化函数、GPIO输出函数、读取GPIO输入的函数。

(1) GPIO模块初始化函数gpio_init：该函数使能GPIO模块的所有输出端口，禁止GPIO模块的所有中断，然后设置GPIO的输出为

0x0f0f0f0f，表示GPIO模块初始化完毕，最后，通过UART输出GPIO模块初始化完毕信息“GPIO initialize done !”。

(2) GPIO输出函数gpio_out：直接将输出值赋给GPIO模块的输出寄存器GPIO_OUT即可。

(3) 读取GPIO输入的函数gpio_in：直接读取GPIO模块的输入寄存器GPIO_IN即可。

第四段：定义了定时器初始化函数OSInitTick。 μ C/OS-II在每个时钟节拍Tick发生一次中断，而每秒发生Tick的次数等于宏OS_TICKS_PER_SEC，这个宏是在文件os_cfg.h中定义的，默认是100，也就是每秒发生100次Tick，每两次Tick相隔10ms，由此就可以设置Compare寄存器的初始值，通过如下式子计算。

系统时钟频率/OS_TICKS_PER_SEC

在函数OSInitTick中，就把通过上式计算得到的值设置为Compare寄存器的初始值。除此之外，还设置Count寄存器为0，因为系统加电后，Count寄存器的值就一直在递增，所以此处将其清零，以重新开始计数。最后，设置Status寄存器的值，使能时钟中断。

第五段：定义了用户任务TaskStart。在15.5.5节介绍过： μ C/OS-II先调用系统初始化函数OSInit，再调用系统启动函数OSStart，在调用OSStart之后做的第一件事就是允许时钟节拍中断。实际上，一般将允许时钟节拍中断的过程放在第一个任务中。所以用户任务TaskStart首先调用之前定义的定时器初始化函数OSInitTick，在其中使能时钟中断。

然后输出Info字符串中的两个字节（对应一个汉字）。设置GPIO模块的输出为count的值。将count加2，等待10个Tick，大约是100ms，再输出Info中的下一个汉字，同时更新GPIO的输出值。读者需要注意一点；在我们上一章建立的SOPC上运行该任务时，4个7段数码管并不是直观显示count的值，比如：count的值为0x00000002，GPIO模块的输出也为0x00000002，但是4个7段数码管上的显示并不是直观的0x00000002，而是如图15-14所示，读者可以参考图14-13给出的7段数码管的引脚与数码管的对应关系进行理解。

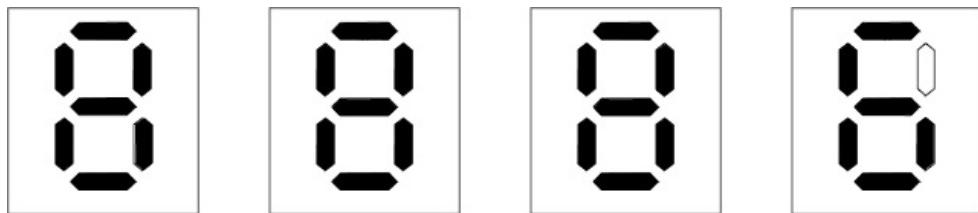


图15-14 count等于0x00000002时的数码管显示

第六段：主函数，这是μC/OS-II中标准的主函数，首先调用操作系统初始化函数OSInit，然后调用各种硬件设备的初始化函数，包括UART控制器的初始化函数、GPIO模块的初始化函数，接着调用函数OSTaskCreate创建一个用户任务，创建的任务就是第五段定义的TaskStart，最后，调用函数OSStart，启动μC/OS-II。

15.13 编译指示文件的建立

经过15.11、15.12两节的工作，我们已经移植μC/OS-II到OpenMIPS处理器，并且编写了测试程序，现在可以建立编译指示文件了，最终目的是通过一条简单的make all命令可以编译所有的代码，得到μC/OS-II的二进制文件。

1. 新建链接脚本文件ram.Id

链接脚本文件ram.ld放在ucosii_OpenMIPS目录下。主要内容如下。

```
MEMORY
{
    vectors : ORIGIN = 0x00000000, LENGTH = 0x00000080

    ram      : ORIGIN = 0x000080, LENGTH = 0x00200000 - 0x00000080
}

SECTIONS
{
    .vectors :
    {
        *(.vectors)
    } > vectors
```

```
.text : {
    *(.text)
    _endtext = .;
} > ram

.rodata : {
    *(.rodata);
    *(.rodata.*)
} > ram

.sbss : {
    *(.sbss)
} > ram

.scommon : {
    *(.scommon)
} > ram

.data : {
    sdata = .;
    _sdata = .;
    *(.data)
    *(.sdata)
    edata = .;
```

```
_edata = .;  
} > ram  
  
.bss SIZEOF(.data) + ADDR(.data) :  
{  
    sbss = . ;  
    _sbss = . ;  
    __bss_start = ALIGN(0x8);  
    ___bss_start = ALIGN(0x8);  
    *(.bss)  
    *(COMMON)  
    end = ALIGN(0x8);  
    _end = ALIGN(0x8);  
    __end = ALIGN(0x8);  
    ebss = . ;  
    _ebss = . ;  
} > ram  
  
.stack :  
{  
    *(.stack)  
  
_stack_addr = .; /* 回顾15.11.3节中，复位异常处理例程入口处的代码 */
```

```
        } > ram  
}
```

在其中定义了很多的Section，这些都是在编译的时候会生成的Section。此外，单独声明一个vectors Section，占用低0x80字节的空间，用来存放异常处理例程入口地址，其余的可执行程序放在地址0x80以上的空间。

ram.ld的最后定义了变量_stack_addr，其值等于堆栈的栈顶地址。该变量在15.11.3节介绍的文件os_cpu_a.S中的复位异常处理例程入口处的代码中使用到。

2. 新建config.mk、Makefile文件

这两个文件都放在ucosii_OpenMIPS目录下。文件config.mk的内容如下，其中定义了一些变量，这些变量对Makefile文件是有用的，包括：编译参数CFLAGS、汇编参数ASFLAGS、链接参数LDFLAGS。注意在CFLAGS的定义中有“-mips32”选项，表示要求编译器按照MIPS32指令集架构（Release 1）编译源程序。

```
#####  
####  
  
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \  
                      else if [ -x /bin/bash ]; then echo  
/bin/bash; \  
                      else echo sh; fi ; fi)
```

```
HOSTCC  = cc

HOSTCFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer

#####
#####

#  

#  Include the make variables (CC, etc...)  

#  

AS      = $(CROSS_COMPILE)as  

LD      = $(CROSS_COMPILE)ld  

CC      = $(CROSS_COMPILE)gcc  

AR      = $(CROSS_COMPILE)ar  

NM      = $(CROSS_COMPILE)nm  

STRIP   = $(CROSS_COMPILE)strip  

OBJCOPY  = $(CROSS_COMPILE)objcopy  

OBJDUMP = $(CROSS_COMPILE)objdump  

RANLIB   = $(CROSS_COMPILE)ranlib  

CFLAGS      += -I$(TOPDIR)/include      -I$(TOPDIR)/ucos      -  

I$(TOPDIR)/common -Wall  

-Wstrict-prototypes -Werror-implicit-function-declaration  

-fomit-frame-pointer -fno-strength-reduce -O2 -g -pipe -fno-  

builtin -nostdlib  

-mips32
```

```
ASFLAGS += $(CFLAGS)

LDFLAGS += -lgcc -e 256

#####
#####

export CONFIG_SHELL HOSTCC HOSTCFLAGS CROSS_COMPILE \
AS LD CC AR NM STRIP OBJCOPY OBJDUMP \
MAKE CFLAGS ASFLAGS

#####
#####

%.o: %.S
$(CC) $(CFLAGS) -c -o $@ $<

%.o: %.C
$(CC) $(CFLAGS) -c -o $@ $<

#####
```

文件Makefile的内容如下，也是位于ucosii_OpenMIPS目录下。

```
ifndef CROSS_COMPILE
CROSS_COMPILE = mips-sde-elf-
endif

export CROSS_COMPILE

#####
#####

# 得到Makefile文件所在的路径
TOPDIR := $(shell if [ "$$PWD" != "" ]; then echo $$PWD; else
pwd; fi)
export TOPDIR

# 引用config.mk文件
include $(TOPDIR)/config.mk

# 定义了几个子目录
SUBDIRS = common ucos port

# 每个子目录下面存在的LIB库文件
LIBS = common/common.o ucos/ucos.o port/port.o

#####
#####

# 编译的目标，注意其中的OS.bin
```

```
all: ucosii.om ucosii.bin ucosii.asm OS.bin

ucosii.om: depend subdirs $(LIBS) Makefile
           $(CC) -Tram.ld -o $@ $(LIBS) -nostdlib $(LDFLAGS)

ucosii.bin: ucosii.om
            mips-sde-elf-objcopy -O binary $< $@

OS.bin: ucosii.bin

./BinMerge.exe -f $< -o $@

ucosii.asm: ucosii.om
            mips-sde-elf-objdump -D $< > $@

#####
#####

depend dep:
```

```

        @for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir .depend ;
done

subdirs:

        @for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir || exit 1
; done

clean:

        find . -type f \
                \(
                    -name 'core' -o -name '*.bak' -o -name '*~'
\

                    -o -name '*.o' -o -name '*.tmp' -o -name
'* .hex' \
                    -o -name 'OS.bin' -o -name 'ucosii.bin' -o -
name '*.srec' \
                    -o -name '*.mem' -o -name '*.img' -o -name
'* .out' \
                    -o -name '*.aux' -o -name '*.log' -o -name
'* .data' \) -print \
                | xargs rm -f
        rm -f System.map

distclean: clean

        find . -type f \
                \(
                    -name .depend -o -name '*.srec' -o -name
'* .bin' \
                    -o -name '*.pdf' \) \

```

```
-print | xargs rm -f  
rm -f $(OBJS) *.bak tags TAGS  
rm -fr *.*~  
  
#####  
#####
```

最后编译得到的μC/OS-II的二进制文件是ucosii.bin，但是此处增加了一步，使用程序MergeBin.exe将ucosii.bin与BootLoader.bin按照图14-24所示的格式合并成一个二进制文件OS.bin。这样只需要将OS.bin写入DE2板上的Flash，当OpenMIPS运行时，会首先运行BootLoader，后者将μC/OS-II的代码复制到SDRAM中，然后跳转到SDRAM，把控制权交给μC/OS-II，于是μC/OS-II就运行起来了。

从上面的Makefile还可以发现，需要在各个子目录下生成对应的LIB库文件，比如：common目录下要有LIB文件common.o，ucos目录下要有LIB文件ucos.o、port目录下要有LIB文件port.o，这些都需要在各个子目录下编写Makefile文件。

3. 在Common目录下添加Makefile文件

其内容如下。LIB文件定义为common.o。

```
# CFLAGS += -DET_DEBUG -DDEBUG  
  
LIB = common.o  
  
OBJS = openmips.o
```

```
all: $(LIB)

$(LIB): $(OBJS) $(SOBJS)
        $(LD) -r -o $@ $(OBJS) $(SOBJS)

#####
#####

.depend:      Makefile $(OBJS:.o=.c)
               $(CC) -M $(CFLAGS) $(OBJS:.o=.c) > $@

sinclude .depend

#####
```

4. 在ucos目录下添加Makefile文件

内容如下。注意在其中的OBJS添加ucos目录下所有的C文件。LIB文件定义为ucos.o。

```
# CFLAGS += -DET_DEBUG -DDEBUG

LIB = ucos.o

OBJS = os_flag.o os_mbox.o os_mem.o os_mutex.o os_q.o os_sem.o
       os_task.o
```

```
os_time.o os_tmr.o os_dbg_r.o os_core.o  
all: $(LIB)  
  
$(LIB): $(OBJS) $(SOBJJS)  
        $(LD) -r -o $@ $(OBJS) $(SOBJJS)  
  
#####  
#####  
  
.depend: Makefile $(OBJS:.o=.c)  
        $(CC) -M $(CFLAGS) $(OBJS:.o=.c) > $@  
  
sinclde .depend  
  
#####  
#####
```

5. 在port目录下添加Makefile文件

内容如下。注意在其中的OBJS添加port目录下的C文件， SOBJJS添加port目录下的汇编文件。LIB文件定义为port.o。

```
LIB    = port.o  
OBJS   = os_cpu_c.o  
SOBJJS = os_cpu_a.o  
  
all: $(LIB)
```

```
$(LIB): $(OBJS) $(SOBJS)
        $(LD) -r -o $@ $(OBJS) $(SOBJS)

#####
#####

.depend: Makefile $(OBJS:.o=.c) $(SOBJS:.o=.S)
          $(CC) -M $(CFLAGS) $(OBJS:.o=.c) $(SOBJS:.o=.S)
> $@

sinclude .depend

#####
#####
```

上述文件建立后，整个μC/OS-II的文件目录就完善了，如图15-15所示。其中加粗斜体的文件都是新增加的文件。

在Ubuntu虚拟机中，打开终端，调整到μC/OS-II工程目录ucosii_OpenMIPS，输入make all，即可得到最终的二进制文件OS.bin，该文件可以直接写入DE2上的Flash。

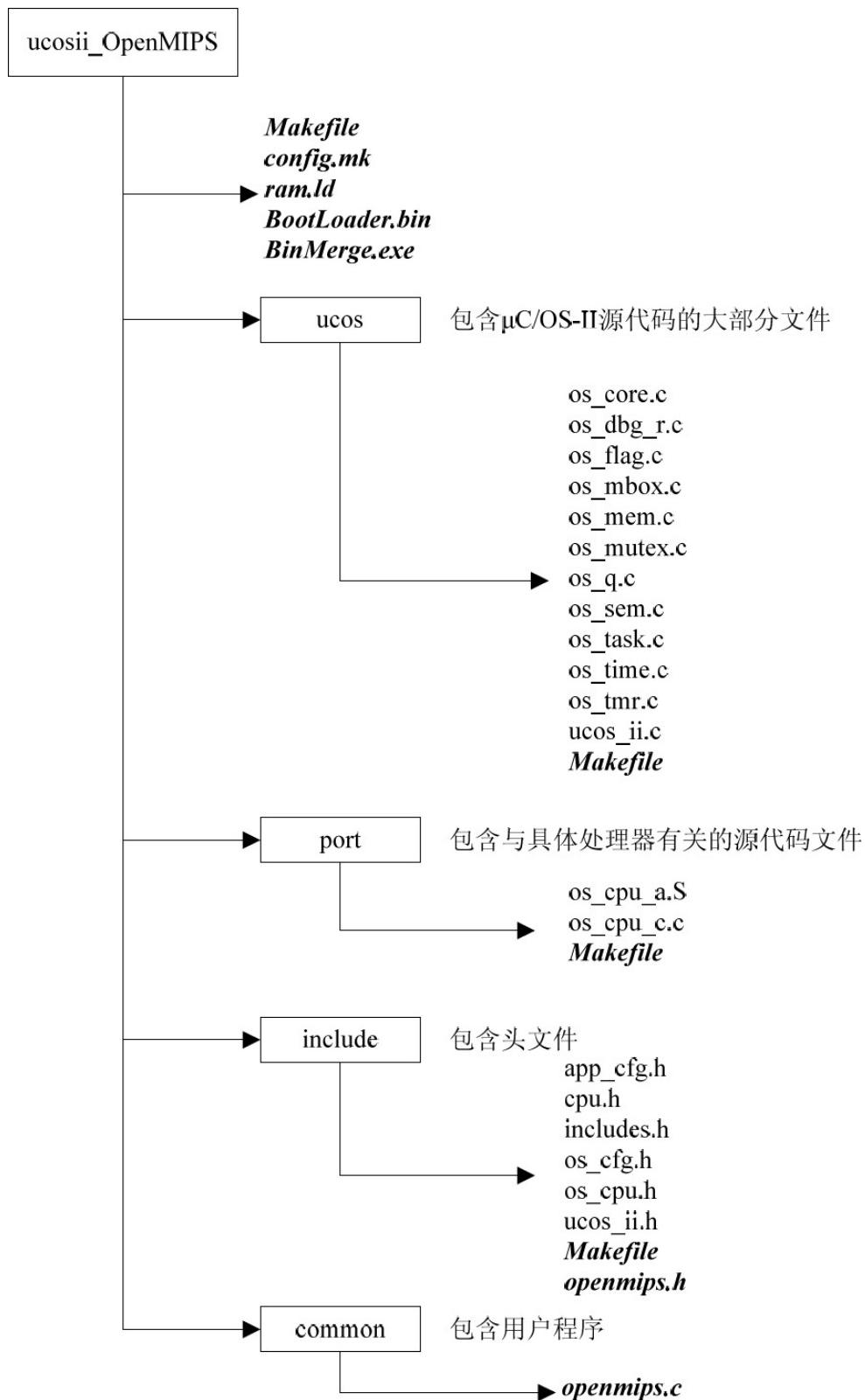


图15-15 μC/OS-II移植工程的整个文件目录

15.14 OpenMIPS处理器运行移植后的μC/OS-II

按照第14章介绍的步骤将文件OS.bin写入DE2开发平台上的Flash，打开PC上的串口程序，将参数设置为与小型SOPC的一样，即设置波特率为9600bps、8位数据位、没有奇偶校验位、1位停止位。通过拨动开关SW17复位OpenMIPS，然后启动OpenMIPS，串口程序会得到如图15-16所示的结果，其中的汉字大概每隔100ms显示一个，另外，4个7段数码管的显示大概每隔100ms变化一次。由此可知，BootLoader加载μC/OS-II成功，μC/OS-II工作正常、移植成功。

读者注意一点，图中显示的字符格式与预期的并不一样，有些地方没有换行，这与串口程序有一定关系，笔者使用超级终端的时候就显示正常。

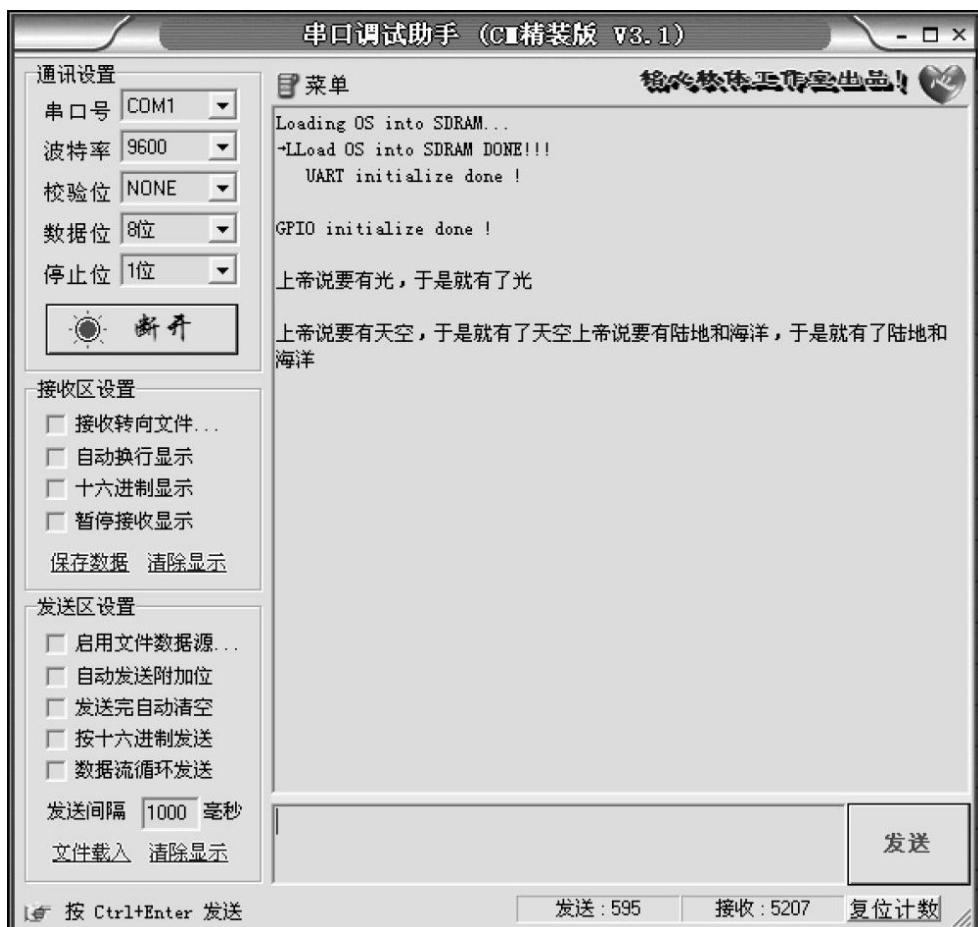


图15-16 μC/OS-II运行结果

15.15 本章小结

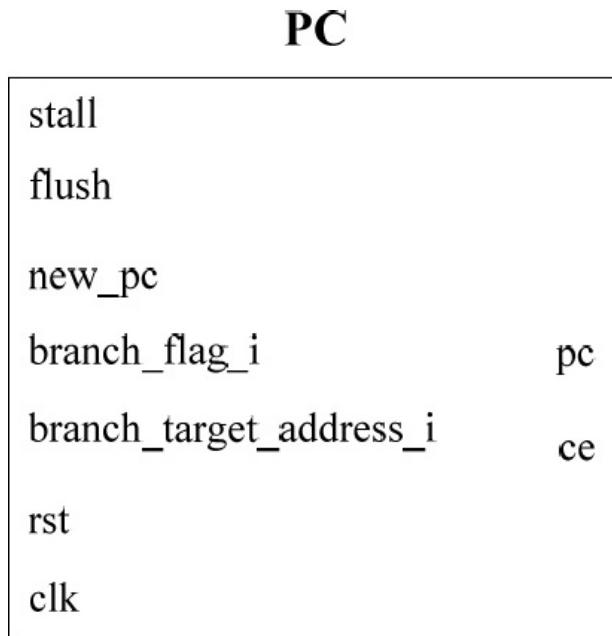
这是本书的最后一章，也是本书最长的一章，其中介绍了μC/OS-II的基本情况、特点、功能等，然后通过修改代码，移植μC/OS-II到OpenMIPS处理器，并且编写测试程序、建立编译指示文件，得到可以在OpenMIPS处理器上运行的μC/OS-II二进制文件。最后通过DE2提供的环境，在OpenMIPS处理器上实际运行μC/OS-II，经过测试，μC/OS-II运行正常，说明我们移植工作的正确性。本章内容涉及的知识比较多，既有操作系统，又有MIPS函数调用规范，还有一些与编译有关的

知识，并且同时使用了汇编和C语言进行程序设计，读者理解起来可能有一定难度，需仔细体会。

附录A 教学版OpenMIPS各个模块的接口说明

A.1 PC模块接口说明

PC模块接口如图A-1所示，采用左边是输入接口，右边是输出接口的方式绘制，目的是便于理解，附录A中其余模块的接口图也都是采用这种方法绘制，后面不再重复说明。各接口的描述如表A-1所示。



pc_reg.v

图A-1 PC模块的外部接口

表A-1 PC模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	branch_flag_i	1	输入	是否发生转移
4	branch_target_address_i	32	输入	转移到的目标地址
5	flush	1	输入	流水线清除信号
6	new_pc	32	输入	异常处理例程入口地址
7	stall	1	输入	取指地址 PC 是否保持不变
8	pc	32	输出	要读取的指令地址
9	ce	1	输出	指令存储器 ROM 使能信号

A.2 IF/ID模块接口说明

IF/ID模块接口如图A-2所示，各接口的描述如表A-2所示。

表A-2 IF/ID模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	if_pc	32	输入	取指阶段取出的指令对应的地址
4	if_inst	32	输入	取指阶段取出的指令
5	stall	1	输入	取指阶段是否暂停
6	flush	1	输入	流水线清除信号
7	id_pc	32	输出	译码阶段的指令对应的地址
8	id_inst	32	输出	译码阶段的指令

A.3 ID模块接口说明

ID模块接口如图A-3所示，各接口的描述如表A-3所示。

IF		ID
stall		aluop_o
flush		alusel_o
if_pc	id_pc	reg1_o
if_inst	id_inst	reg2_o
rst		wd_o
clk		wreg_o
if_id.v		
		excepttype_o
		inst_o
		current_inst_address_o
		is_in_delayslot_o
		link_addr_o
		next_inst_in_delayslot_o
	is_in_delayslot_i	reg1_read_o
		reg1_addr_o
		reg2_read_o
	rst	reg2_addr_o
		stallreq
	reg1_data_i	branch_flag_o
	reg2_data_i	branch_target_address_o
id.v		

图A-2 IF/ID模块的外部接口

图A-3 ID模块的外部接口

表A-3 ID模块的接口描述

序号	接口名	宽度 (bit)	输入/ 输出	作用
1	rst	1	输入	复位信号
2	pc_i	32	输入	译码阶段的指令对应的地址
3	inst_i	32	输入	译码阶段的指令
4	reg1_data_i	32	输入	从 Regfile 输入的第一个读寄存器端口的输入
5	reg2_data_i	32	输入	从 Regfile 输入的第二个读寄存器端口的输入
6	ex_wreg_i	1	输入	处于执行阶段的指令是否要写目的寄存器
7	ex_wd_i	5	输入	处于执行阶段的指令要写的目的寄存器地址
8	ex_wdata_i	32	输入	处于执行阶段的指令要写入目的寄存器的数据
9	mem_wreg_i	1	输入	处于访存阶段的指令是否要写目的寄存器
10	mem_wd_i	5	输入	处于访存阶段的指令要写的目的寄存器地址
11	mem_wdata_i	32	输入	处于访存阶段的指令要写入目的寄存器的数据
12	ex_aluop_i	8	输入	处于执行阶段指令的运算子类型
13	is_in_delayslot_i	1	输入	当前处于译码阶段的指令是否位于延迟槽
14	reg1_read_o	1	输出	Regfile 模块的第一个读寄存器端口的读使能信号
15	reg2_read_o	1	输出	Regfile 模块的第二个读寄存器端口的读使能信号
16	reg1_addr_o	5	输出	Regfile 模块的第一个读寄存器端口的读地址信号
17	reg2_addr_o	5	输出	Regfile 模块的第二个读寄存器端口的读地址信号
18	aluop_o	8	输出	译码阶段的指令要进行的运算的子类型
19	alusel_o	3	输出	译码阶段的指令要进行的运算的类型
20	reg1_o	32	输出	译码阶段的指令要进行的运算的源操作数 1
21	reg2_o	32	输出	译码阶段的指令要进行的运算的源操作数 2
22	wd_o	5	输出	译码阶段的指令要写入的目的寄存器地址
23	wreg_o	1	输出	译码阶段的指令是否有要写入的目的寄存器
24	branch_flag_o	1	输出	是否发生转移
25	branch_target_address_o	32	输出	转移到的目标地址
26	is_in_delayslot_o	1	输出	当前处于译码阶段的指令是否位于延迟槽
27	link_addr_o	32	输出	转移指令要保存的返回地址
28	next_inst_in_delayslot_o	1	输出	下一条进入译码阶段的指令是否位于延迟槽
29	inst_o	32	输出	当前处于译码阶段的指令
30	excepttype_o	32	输出	收集的异常信息
31	current_inst_addr_o	32	输出	译码阶段指令的地址
32	stallreq	1	输出	译码阶段请求流水暂停

A.4 Regfile模块接口说明

Regfile模块接口如图A-4所示，各接口的描述如表A-4所示。

表A-4 Regfile模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号, 高电平有效
2	clk	1	输入	时钟信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据
5	we	1	输入	写使能信号
6	raddr1	5	输入	第一个读寄存器端口要读取的寄存器的地址
7	re1	1	输入	第一个读寄存器端口读使能信号
8	rdata1	32	输出	第一个读寄存器端口输出的寄存器值
9	raddr2	5	输入	第二个读寄存器端口要读取的寄存器的地址
10	re2	1	输入	第二个读寄存器端口读使能信号
11	rdata2	32	输出	第二个读寄存器端口输出的寄存器值

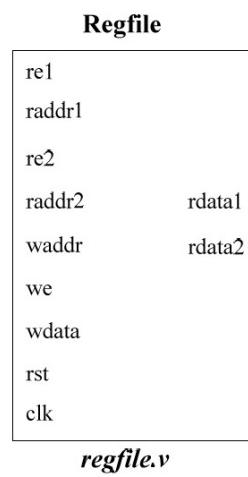
A.5 ID/EX模块接口说明

ID/EX模块接口如图A-5所示，各接口的描述如表A-5所示。

ID/EX

stall	ex_aluop
flush	ex_alusel
id_aluop	ex_reg1
id_alusel	ex_reg2
id_reg1	ex_wd
id_reg2	ex_wreg
id_wd	ex_excepttype
id_wreg	ex_link_address
id_excepttype	ex_inst
id_inst	ex_is_in_delayslot
id_current_inst_address	
id_is_in_delayslot	ex_current_inst_address
id_link_address	
next_inst_in_delayslot_i	
rst	is_in_delayslot_o
clk	

id_ex.v



图A-4 Regfile模块的外部接口

图A-5 ID/EX模块的外部接口

表A-5 ID/EX模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	id_alusel	3	输入	译码阶段的指令要进行的运算的类型
4	id_aluop	8	输入	译码阶段的指令要进行的运算的子类型
5	id_reg1	32	输入	译码阶段的指令要进行的运算的源操作数1
6	id_reg2	32	输入	译码阶段的指令要进行的运算的源操作数2
7	id_wd	5	输入	译码阶段的指令要写入的目的寄存器地址
8	id_wreg	1	输入	译码阶段的指令是否有要写入的目的寄存器
9	stall	1	输入	译码阶段是否暂停
10	flush	1	输入	流水线清除信号
11	id_excepttype	32	输入	译码阶段收集到的异常信息
12	id_current_inst_addr	32	输入	译码阶段指令的地址
13	id_is_in_delayslot	1	输入	当前处于译码阶段的指令是否位于延迟槽
14	id_link_address	32	输入	处于译码阶段的转移指令要保存的返回地址
15	next_inst_in_delayslot_i	1	输入	下一条进入译码阶段的指令是否位于延迟槽
16	id_inst	32	输入	当前处于译码阶段的指令
17	ex_inst	32	输出	当前处于执行阶段的指令
18	ex_is_in_delayslot	32	输出	当前处于执行阶段的指令是否位于延迟槽
19	ex_link_address	1	输出	处于执行阶段的转移指令要保存的返回地址
20	is_in_delayslot_o	1	输出	当前处于译码阶段的指令是否位于延迟槽
21	ex_excepttype	32	输出	译码阶段收集到的异常信息
22	ex_current_inst_addr	32	输出	执行阶段指令的地址
23	ex_alusel	3	输出	执行阶段的指令要进行的运算的类型
24	ex_aluop	8	输出	执行阶段的指令要进行的运算的子类型
25	ex_reg1	32	输出	执行阶段的指令要进行的运算的源操作数1
26	ex_reg2	32	输出	执行阶段的指令要进行的运算的源操作数2
27	ex_wd	5	输出	执行阶段的指令要写入的目的寄存器地址
28	ex_wreg	1	输出	执行阶段的指令是否有要写入的目的寄存器

A.6 EX模块接口说明

EX模块接口如图A-6所示，各接口的描述如表A-6所示。

EX

aluop_i	
alusel_i	
reg1_i	cp0_reg_we_o
reg2_i	cp0_reg_write_addr_o
wd_i	cp0_reg_data_o
wreg_i	mem_addr_o
excepttype_i	
link_address_i	reg2_o
inst_i	hi_o
is_in_delayslot_i	lo_o
current_inst_address_i	whilo_o
hi_i	hi_lo_temp_o
lo_i	cnt_o
cp0_reg_data_i	excepttype_o
wb_hi_i	is_in_delayslot_o
wb_lo_i	current_inst_address_o
wb_whilo_i	aluop_o
wb_cp0_reg_we	wreg_o
wb_cp0_reg_write_addr	wd_o
wb_cp0_reg_data	wdata_o
mem_hi_i	
mem_lo_i	stallreq
mem_whilo_i	cp0_reg_read_addr_o
mem_cp0_reg_we	
mem_cp0_reg_write_addr	signed_div_o
mem_cp0_reg_data	div_opdata1_o
hi_lo_temp_i	div_opdata2_o
cnt_i	div_start_o
div_result_i	
div_ready_i	
rst	

图A-6 EX模块的外部接口

表A-6 EX模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	alusel_i	3	输入	执行阶段要进行的运算的类型
3	aluop_i	8	输入	执行阶段要进行的运算的子类型
4	reg1_i	32	输入	参与运算的源操作数1

续表

序号	接口名	宽度(bit)	输入/输出	作用
5	reg2_i	32	输入	参与运算的源操作数2
6	wd_i	5	输入	指令执行要写入的目的寄存器地址
7	wreg_i	1	输入	是否有要写入的目的寄存器
8	excepttype_i	32	输入	译码阶段收集到的异常信息
9	current_inst_addr_i	32	输入	执行阶段指令的地址
10	is_in_delayslot_i	1	输入	当前处于执行阶段的指令是否位于延迟槽
11	link_address_i	32	输入	处于执行阶段的转移指令要保存的返回地址
12	hi_lo_temp_i	64	输入	第一个执行周期得到的乘法结果
13	cnt_i	2	输入	当前处于执行阶段的第几个时钟周期
14	hi_lo_temp_o	64	输出	第一个执行周期得到的乘法结果
15	cnt_o	2	输出	下一个时钟周期处于执行阶段的第几个时钟周期
16	excepttype_o	32	输出	译码阶段、执行阶段收集到的异常信息
17	current_inst_addr_o	32	输出	执行阶段指令的地址
18	is_in_delayslot_o	1	输出	执行阶段的指令是否是延迟槽指令
19	wd_o	5	输出	执行阶段的指令最终要写入的目的寄存器地址
20	wreg_o	1	输出	执行阶段的指令最终是否有要写入的目的寄存器
21	wdata_o	32	输出	执行阶段的指令最终要写入目的寄存器的值
22	signed_div_o	1	输出	是否是有符号除法,为1表示是有符号除法
23	div_opdata1_o	32	输出	被除数
24	div_opdata2_o	32	输出	除数
25	div_start_o	1	输出	是否开始除法运算
26	div_result_i	64	输入	除法运算结果
27	div_ready_i	1	输入	除法运算是否结束
28	hi_i	32	输入	HILO模块给出的HI寄存器的值
29	lo_i	32	输入	HILO模块给出的LO寄存器的值
30	mem_whilo_i	1	输入	处于访存阶段的指令是否要写HI、LO寄存器
31	mem_hi_i	32	输入	处于访存阶段的指令要写入HI寄存器的值
32	mem_lo_i	32	输入	处于访存阶段的指令要写入LO寄存器的值
33	wb_whilo_i	1	输入	处于回写阶段的指令是否要写HI、LO寄存器
34	wb_hi_i	32	输入	处于回写阶段的指令要写入HI寄存器的值
35	wb_lo_i	32	输入	处于回写阶段的指令要写入LO寄存器的值

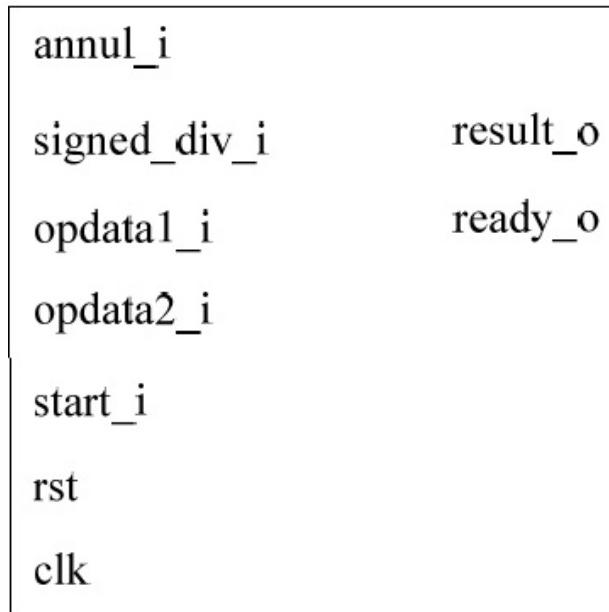
续表

序号	接口名	宽度 (bit)	输入/ 输出	作用
36	whilo_o	1	输出	执行阶段的指令是否要写 HI、LO 寄存器
37	hi_o	32	输出	执行阶段的指令要写入 HI 寄存器的值
38	lo_o	32	输出	执行阶段的指令要写入 LO 寄存器的值
39	cp0_reg_data_i	32	输入	从 CP0 模块读取的指定寄存器的值
40	mem_cp0_reg_we	1	输入	访存阶段的指令是否要写 CP0 中的寄存器
41	mem_cp0_reg_write_addr	5	输入	访存阶段的指令要写的 CP0 中寄存器的地址
42	mem_cp0_reg_data	32	输入	访存阶段的指令要写入 CP0 中寄存器的数据
43	wb_cp0_reg_we	1	输入	回写阶段的指令是否要写 CP0 中的寄存器
44	wb_cp0_reg_write_addr	5	输入	回写阶段的指令要写的 CP0 中寄存器的地址
45	wb_cp0_reg_data	32	输入	回写阶段的指令要写入 CP0 中寄存器的数据
46	cp0_reg_read_addr_o	5	输出	执行阶段的指令要读取的 CP0 中寄存器的地址
47	cp0_reg_we_o	1	输出	执行阶段的指令是否要写 CP0 中的寄存器
48	cp0_reg_write_addr_o	5	输出	执行阶段的指令要写的 CP0 中寄存器的地址
49	cp0_reg_data_o	32	输出	执行阶段的指令要写入 CP0 中寄存器的数据
50	inst_i	32	输入	当前处于执行阶段的指令
51	aluop_o	8	输出	执行阶段的指令要进行的运算的子类型
52	mem_addr_o	32	输出	加载、存储指令对应的存储器地址
53	reg2_o	32	输出	存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值
54	stallreq	1	输出	执行阶段是否请求流水线暂停

A.7 DIV模块接口说明

DIV模块接口如图A-7所示，各接口的描述如表A-7所示。

DIV



div.v

图A-7 DIV模块的外部接口

表A-7 DIV模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号, 高电平有效
2	clk	1	输入	时钟信号
3	signed_div_i	1	输入	是否是有符号除法, 为 1 表示有符号除法
4	opdata1_i	32	输入	被除数
5	opdata2_i	32	输入	除数
6	start_i	1	输入	是否开始除法运算
7	annul_i	1	输入	是否取消除法运算, 为 1 表示取消除法运算
8	result_o	64	输出	除法运算结果
9	ready_o	1	输出	除法运算是否结束

A.8 EX/MEM模块接口说明

EX/MEM模块接口如图A-8所示，各接口的描述如表A-8所示。

EX/MEM

stall	mem_wd
flush	mem_wreg
ex_cp0_reg_we	mem_wdata
ex_cp0_reg_write_addr	mem_hi
ex_cp0_reg_data	mem_lo
ex_mem_addr	mem_whilo
	mem_aluop
ex_reg2	mem_mem_addr
ex_hi	
ex_lo	mem_reg2
ex_whilo	mem_cp0_reg_we
hilo_i	mem_cp0_reg_write_addr
cnt_i	mem_cp0_reg_data
ex_excepttype	mem_excepttype
ex_is_in_delayslot	mem_is_in_delayslot
ex_current_inst_address	mem_current_inst_address
ex_aluop	hilo_o
ex_wreg	cnt_o
ex_wd	
ex_wdata	
rst	
clk	

ex_mem.v

图A-8 EX/MEM模块的外部接口

表A-8 EX/MEM模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	ex_wd	5	输入	执行阶段的指令执行后要写入的目的寄存器地址
4	ex_wreg	1	输入	执行阶段的指令执行后是否有要写入的目的寄存器
5	ex_wdata	32	输入	执行阶段的指令执行后要写入目的寄存器的值
6	mem_wd	5	输出	访存阶段的指令要写入的目的寄存器地址
7	mem_wreg	1	输出	访存阶段的指令是否有要写入的目的寄存器
8	mem_wdata	32	输出	访存阶段的指令要写入目的寄存器的值
9	stall	1	输入	执行阶段是否暂停
10	flush	1	输入	是否清除流水线
11	ex_cp0_reg_we	1	输入	执行阶段的指令是否要写 CP0 中的寄存器
12	ex_cp0_reg_write_addr	5	输入	执行阶段的指令要写的 CP0 中寄存器的地址
13	ex_cp0_reg_data	32	输入	执行阶段的指令要写入 CP0 中寄存器的数据
14	mem_cp0_reg_we	1	输出	访存阶段的指令是否要写 CP0 中的寄存器
15	mem_cp0_reg_write_addr	5	输出	访存阶段的指令要写的 CP0 中寄存器的地址
16	mem_cp0_reg_data	32	输出	访存阶段的指令要写入 CP0 中寄存器的数据
17	ex_aluop	8	输入	执行阶段的指令要进行的运算的子类型
18	ex_mem_addr	32	输入	执行阶段的加载、存储指令对应的存储器地址
19	ex_reg2	32	输入	执行阶段的存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值
20	mem_aluop	8	输出	访存阶段的指令要进行的运算的子类型
21	mem_mem_addr	32	输出	访存阶段的加载、存储指令对应的存储器地址
22	mem_reg2	32	输出	访存阶段的存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值
23	ex_whilo	1	输入	执行阶段的指令是否要写 HI、LO 寄存器
24	ex_hi	32	输入	执行阶段的指令要写入 HI 寄存器的值
25	ex_lo	32	输入	执行阶段的指令要写入 LO 寄存器的值
26	mem_whilo	1	输出	访存阶段的指令是否要写 HI、LO 寄存器
27	mem_hi	32	输出	访存阶段的指令要写入 HI 寄存器的值
28	mem_lo	32	输出	访存阶段的指令要写入 LO 寄存器的值
29	ex_excepttype	32	输入	译码、执行阶段收集到的异常信息
30	ex_current_inst_address	32	输入	执行阶段指令的地址
31	ex_is_in_delayslot	1	输入	执行阶段的指令是否是延迟槽指令
32	mem_excepttype	32	输出	译码、执行阶段收集到的异常信息

续表

序号	接口名	宽度(bit)	输入/输出	作用
33	mem_current_inst_address	32	输出	访存阶段指令的地址
34	mem_is_in_delayslot	1	输出	访存阶段的指令是否是延迟槽指令
35	hilo_i	64	输入	保存的乘法结果
36	cnt_i	2	输入	下一个时钟周期是执行阶段的第几个时钟周期
37	hilo_o	64	输出	保存的乘法结果
38	cnt_o	2	输出	当前处于执行阶段的第几个时钟周期

A.9 MEM模块接口说明

MEM模块接口如图A-9所示，各接口的描述如表A-9所示。

MEM

wd_i	
wreg_i	cp0_epc_o
wdata_i	LLbit_value_o
hi_i	LLbit_we_o
lo_i	wd_o
whilo_i	wreg_o
aluop_i	wdata_o
mem_addr_i	hi_o
	lo_o
reg2_i	whilo_o
cp0_reg_we_i	cp0_reg_we_o
cp0_reg_write_addr_i	cp0_reg_write_addr_o
cp0_reg_data_i	cp0_reg_data_o
excepttype_i	
is_in_delayslot_i	excepttype_o
current_inst_address_i	
LLbit_i	current_inst_address_o
cp0_status_i	is_in_delayslot_o
cp0_cause_i	
cp0_epc_i	
wb_LLbit_we_i	mem_we_o
wb_LLbit_value_i	mem_addr_o
wb_cp0_reg_we	mem_sel_o
wb_cp0_reg_write_addr	mem_data_o
wb_cp0_reg_data	mem_ce_o
mem_data_i	
rst	

图A-9 MEM模块的外部接口

表A-9 MEM模块的接口描述

序号	接口名	宽度(bit)	输入/输出	作用
1	rst	1	输入	复位信号
2	wd_i	5	输入	访存阶段的指令要写入的目的寄存器地址
3	wreg_i	1	输入	访存阶段的指令是否有要写入的目的寄存器
4	wdata_i	32	输入	访存阶段的指令要写入目的寄存器的值
5	wd_o	5	输出	访存阶段的指令最终要写入的目的寄存器地址
6	wreg_o	1	输出	访存阶段的指令最终是否有要写入的目的寄存器
7	wdata_o	32	输出	访存阶段的指令最终要写入目的寄存器的值
8	aluop_i	8	输入	访存阶段的指令要进行的运算的子类型
9	mem_addr_i	32	输入	访存阶段的加载、存储指令对应的存储器地址
10	reg2_i	32	输入	访存阶段的存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值
11	mem_data_i	32	输入	从数据存储器读取的数据
12	mem_addr_o	32	输出	要访问的数据存储器的地址
13	mem_we_o	1	输出	是否是写操作，为 1 表示是写操作
14	mem_sel_o	4	输出	字节选择信号
15	mem_data_o	32	输出	要写入数据存储器的数据
16	mem_ce_o	1	输出	数据存储器使能信号
17	whilo_i	1	输入	访存阶段的指令是否要写 HI、LO 寄存器
18	hi_i	32	输入	访存阶段的指令要写入 HI 寄存器的值
19	lo_i	32	输入	访存阶段的指令要写入 LO 寄存器的值
20	whilo_o	1	输出	访存阶段的指令最终是否要写 HI、LO 寄存器
21	hi_o	32	输出	访存阶段的指令最终要写入 HI 寄存器的值
22	lo_o	32	输出	访存阶段的指令最终要写入 LO 寄存器的值
23	cp0_reg_we_i	1	输入	访存阶段的指令是否要写 CP0 中的寄存器
24	cp0_reg_write_addr_i	5	输入	访存阶段的指令要写的 CP0 中寄存器的地址
25	cp0_reg_data_i	32	输入	访存阶段的指令要写入 CP0 中寄存器的数据
26	cp0_reg_we_o	1	输出	访存阶段的指令最终是否要写 CP0 中的寄存器
27	cp0_reg_write_addr_o	5	输出	访存阶段的指令最终要写的 CP0 中寄存器的地址
28	cp0_reg_data_o	32	输出	访存阶段的指令最终要写入 CP0 中寄存器的数据
29	excepttype_i	32	输入	译码、执行阶段收集到的异常信息
30	current_inst_address_i	32	输入	访存阶段指令的地址
31	is_in_delayslot_i	1	输入	访存阶段的指令是否是延迟槽指令
32	cp0_status_i	32	输入	CP0 中 Status 寄存器的值
33	cp0_cause_i	32	输入	CP0 中 Cause 寄存器的值

续表

序号	接口名	宽度(bit)	输入/输出	作用
34	cp0_epc_i	32	输入	CP0 中 EPC 寄存器的值
35	wb_cp0_reg_we	1	输入	回写阶段的指令是否要写 CP0 中的寄存器
36	wb_cp0_reg_write_address	5	输入	回写阶段的指令要写的 CP0 中寄存器的地址
37	wb_cp0_reg_data	32	输入	回写阶段的指令要写入 CP0 中寄存器的值
38	excepttype_o	32	输出	最终的异常类型
39	current_inst_address_o	32	输出	访存阶段指令的地址
40	is_in_delayslot_o	1	输出	访存阶段的指令是否是延迟槽指令
41	cp0_enc_o	32	输出	CP0 中 EPC 寄存器的最新值

42	LLbit_i	1	输入	LLbit 模块给出的 LLbit 寄存器的值
43	wb_LLbit_we_i	1	输入	回写阶段的指令是否要写 LLbit 寄存器
44	wb_LLbit_value_i	1	输入	回写阶段要写入 LLbit 寄存器的值
45	LLbit_we_o	1	输出	访存阶段的指令是否要写 LLbit 寄存器
46	LLbit_value_o	1	输出	访存阶段的指令要写入 LLbit 寄存器的值

A.10 MEM/WB模块接口说明

MEM/WB模块接口如图A-10所示，各接口的描述如表A-10所示。

MEM/WB

stall	wb_whilo
flush	wb_hi
mem_LLbit_value	wb_lo
mem_LLbit_we	wb_LLbit_we
mem_wd	wb_LLbit_value
mem_wreg	
mem_wdata	wb_wd
mem_hi	wb_wreg
mem_lo	wb_wdata
mem_whilo	wb_cp0_reg_we
mem_cp0_reg_we	wb_cp0_reg_write_addr
mem_cp0_reg_write_addr	wb_cp0_reg_data
mem_cp0_reg_data	
rst	
clk	

mem_wb.v

图A-10 MEM/WB模块的外部接口

表A-10 MME/WB模块的接口描述

序 号	接 口 名	宽 度 (bit)	输入/ 输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	mem_wd	5	输入	访存阶段的指令最终要写入的目的寄存器地址
4	mem_wreg	1	输入	访存阶段的指令最终是否有要写入的目的寄存器
5	mem_wdata	32	输入	访存阶段的指令最终要写入目的寄存器的值
6	wb_wd	5	输出	回写阶段的指令要写入的目的寄存器地址
7	wb_wreg	1	输出	回写阶段的指令是否有要写入的目的寄存器
8	wb_wdata	32	输出	回写阶段的指令要写入目的寄存器的值
9	mem_LLbit_we	1	输入	访存阶段的指令是否要写 LLbit 寄存器
10	mem_LLbit_value	1	输入	访存阶段的指令要写入 LLbit 寄存器的值
11	wb_LLbit_we	1	输出	回写阶段的指令是否要写 LLbit 寄存器
12	wb_LLbit_value	1	输出	回写阶段的指令要写入 LLbit 寄存器的值
13	mem_cp0_reg_we	1	输入	访存阶段的指令是否要写 CP0 中的寄存器
14	mem_cp0_reg_write_addr	5	输入	访存阶段的指令要写的 CP0 中寄存器的地址
15	mem_cp0_reg_data	32	输入	访存阶段的指令要写入 CP0 中寄存器的数据
16	wb_cp0_reg_we	1	输出	回写阶段的指令是否要写 CP0 中的寄存器
17	wb_cp0_reg_write_addr	5	输出	回写阶段的指令要写的 CP0 中寄存器的地址
18	wb_cp0_reg_data	32	输出	回写阶段的指令要写入 CP0 中寄存器的数据
19	mem_whilo	1	输入	访存阶段的指令是否要写 HI、LO 寄存器
20	mem_hi	32	输入	访存阶段的指令要写入 HI 寄存器的值
21	mem_lo	32	输入	访存阶段的指令要写入 LO 寄存器的值
22	wb_whilo	1	输出	回写阶段的指令是否要写 HI、LO 寄存器
23	wb_hi	32	输出	回写阶段的指令要写入 HI 寄存器的值
24	wb_lo	32	输出	回写阶段的指令要写入 LO 寄存器的值
25	stall	1	输入	访存阶段是否暂停
26	flush	1	输入	是否清除流水线

A.11 CP0模块接口说明

CP0模块接口如图A-11所示，各接口的描述如表A-11所示。

CP0

we_i	data_o
waddr_i	count_o
data_i	compare_o
excepttype_i	status_o
int_i	cause_o
current_inst_addr_i	epc_o
is_in_delayslot_i	config_o
raddr_i	prid_o
rst	
clk	timer_int_o

cp0_reg.v

图A-11 CP0模块的外部接口

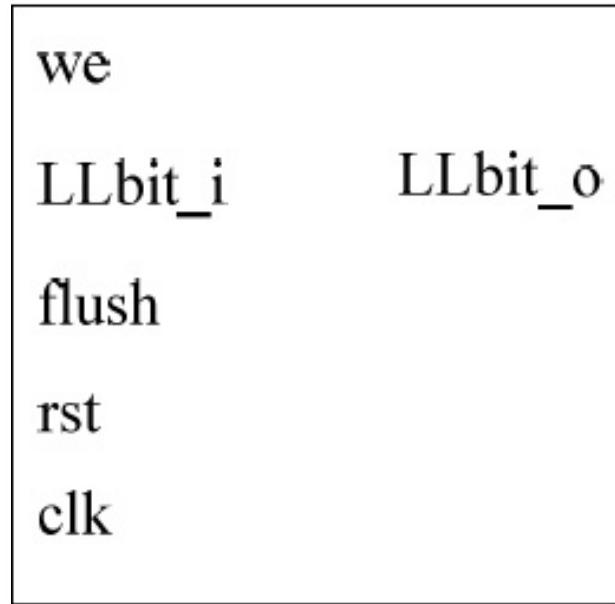
表A-11 CP0模块的接口描述

序 号	接 口 名	宽 度 (bit)	输入/ 输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	raddr_i	5	输入	要读取的 CP0 中寄存器的地址
4	int_i	6	输入	6 个外部硬件中断输入
5	we_i	1	输入	是否要写 CP0 中的寄存器
6	waddr_i	5	输入	要写的 CP0 中寄存器的地址
7	wdata_i	32	输入	要写入 CP0 中寄存器的数据
8	data_o	32	输出	读出的 CP0 中某个寄存器的值
9	count_o	32	输出	Count 寄存器的值
10	compare_o	32	输出	Compare 寄存器的值
11	status_o	32	输出	Status 寄存器的值
12	cause_o	32	输出	Cause 寄存器的值
13	epc_o	32	输出	EPC 寄存器的值
14	config_o	32	输出	Config 寄存器的值
15	prid_o	32	输出	PRId 寄存器的值
16	timer_int_o	1	输出	是否有定时中断发生
17	excepttype_o	32	输入	最终的异常类型
18	current_inst_address_o	32	输入	发生异常的指令地址
19	is_in_delayslot_o	1	输入	发生异常的指令是否是延迟槽指令

A.12 LLbit模块接口说明

LLbit模块接口如图A-12所示，各接口的描述如表A-12所示。

LLbit



LLbit_reg.v

图A-12 LLbit模块的外部接口

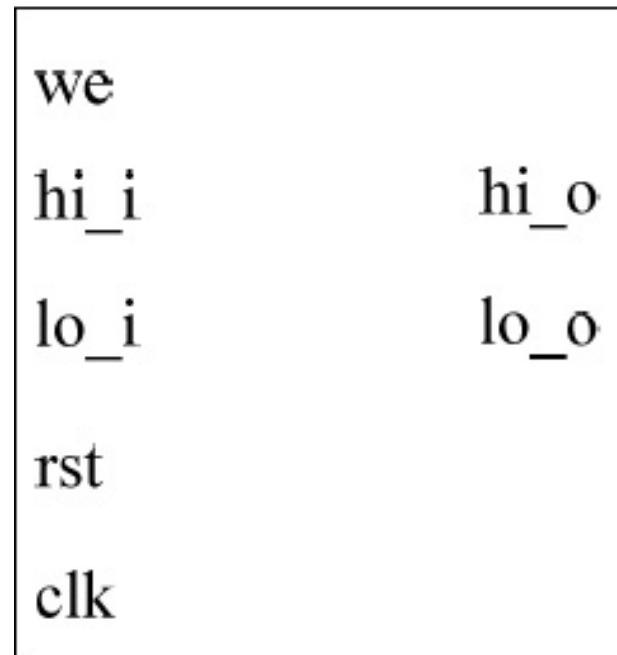
表A-12 LLbit模块的接口描述

序号	接 口 名	宽 度 (bit)	输入/ 输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	flush	1	输入	是否有异常发生
4	we	1	输入	是否要写 LLbit 寄存器
5	LLbit_i	1	输入	要写入 LLbit 寄存器的值
6	LLbit_o	1	输出	LLbit 寄存器的值

A.13 HILo模块接口说明

HILo模块接口如图A-13所示，各接口的描述如表A-13所示。

HILO



hilo_reg.v

图A-13 HILO模块的外部接口

表A-13 HILO模块的接口描述

序号	接 口 名	宽度 (bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号

续表

序号	接 口 名	宽度 (bit)	输入/输出	作 用
3	we	1	输入	HI、LO 寄存器写使能信号
4	hi_i	32	输入	要写入 HI 寄存器的值
5	lo_i	32	输入	要写入 LO 寄存器的值
6	hi_o	32	输出	HI 寄存器的值
7	lo_o	32	输出	LO 寄存器的值

A.14 CTRL模块接口说明

CTRL模块接口如图A-14所示，各接口的描述如表A-14所示。

CTRL

excepttype_i	new_pc
cp0_epc_i	stall
stallreq_from_ex	flush
stallreq_from_id	
rst	

ctrl.v

图A-14 CTRL模块的外部接口

表A-14 CTRL模块的接口描述

序号	接 口 名	宽度(bit)	输入/输出	作 用
1	rst	1	输入	复位信号
2	stallreq_from_id	1	输入	处于译码阶段的指令是否请求流水线暂停信号
3	stallreq_from_ex	1	输入	处于执行阶段的指令是否请求流水线暂停信号
4	stall	6	输出	暂停流水线控制信号
5	cp0_epc_i	32	输入	EPC 寄存器的最新值
6	excepttype_i	32	输入	最终的异常类型
7	new_pc	32	输出	异常处理入口地址
8	flush	1	输出	是否清除流水线

附录B OpenMIPS实现的所有指令及对应的机器码

B.1 逻辑操作指令

31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000	rs	rt	rd	00000	AND 100100		and rd, rs, rt
SPECIAL 000000	rs	rt	rd	00000	OR 100101		or rd, rs, rt
SPECIAL 000000	rs	rt	rd	00000	XOR 100110		xor rd, rs, rt
SPECIAL 000000	rs	rt	rd	00000	NOR 100111		nor rd, rs, rt
ANDI 001100	rs	rt		immediate			andi rt, rs, immediate
XORI 001110	rs	rt		immediate			xori rt, rs, immediate
LUI 001111	00000	rt		immediate			lui rt, immediate
ORI 001101	rs	rt		immediate			ori rs, rt, immediate

B.2 移位操作指令

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000		00000	rt	rd	sa	SLL 000000		sll rd, rt, sa
SPECIAL 000000		00000	rt	rd	sa	SRL 000010		srl rd, rt, sa
SPECIAL 000000		00000	rt	rd	sa	SRA 000011		sra rd, rt, sa
SPECIAL 000000		rs	rt	rd	00000	SLLV 000100		sllv rd, rt, rs
SPECIAL 000000		rs	rt	rd	00000	SRLV 000110		srlv rd, rt, rs
SPECIAL 000000		rs	rt	rd	00000	SRAV 000111		srv rd, rt, rs

B.3 移动操作指令

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000		rs	rt	rd	00000	MOVN 001011		movn rd, rs, rt
SPECIAL 000000		rs	rt	rd	00000	MOVZ 001010		movz rd, rs, rt
SPECIAL 000000		00000	00000	rd	00000	MFHI 010000		mfhi rd
SPECIAL 000000		00000	00000	rd	00000	MFLO 010010		mflo rd
SPECIAL 000000		rs	00000	00000	00000	MTHI 010001		mthi rs
SPECIAL 000000		rs	00000	00000	00000	MTLO 010011		mtlo rs

B.4 算术操作指令

31	26	25	21	20	16	15	11	10	6	5	0	
SPECIAL 000000		rs		rt		rd		00000		ADD 100000		add rd, rs, rt
SPECIAL 000000		rs		rt		rd		00000		ADDU 100001		addu rd, rs, rt
SPECIAL 000000		rs		rt		rd		00000		SUB 100010		sub rd, rs, rt
SPECIAL 000000		rs		rt		rd		00000		SUBU 100011		subu rd, rs, rt
SPECIAL 000000		rs		rt		rd		00000		SLT 101010		slt rd, rs, rt
SPECIAL 000000		rs		rt		rd		00000		SLTU 101011		sltu rd, rs, rt
SPECIAL 000000		rs		rt		00000		00000		MULT 011000		mult rs, st
SPECIAL 000000		rs		rt		00000		00000		MULTU 011001		multu rs, st
SPECIAL 000000		rs		rt		00000		00000		DIV 011010		div rs, rt
SPECIAL 000000		rs		rt		00000		00000		DIVU 011011		divu rs, rt
SPECIAL2 011100		rs		rt		00000		00000		MADD 000000		madd rs, rt
SPECIAL2 011100		rs		rt		00000		00000		MADDU 000001		maddu rs, rt
SPECIAL2 011100		rs		rt		00000		00000		MSUB 000100		msub rs, rt
SPECIAL2 011100		rs		rt		00000		00000		MSUBU 000101		msubu rs, rt
SPECIAL2 011100		rs		rt		rd		00000		CLZ 100000		clz rd, rs
SPECIAL2 011100		rs		rt		rd		00000		CLO 100001		clo rd, rs
SPECIAL2 011100		rs		rt		rd		00000		MUL 000010		mul rd, rs, st
ADDI 001000		rs		rt				immediate				addi rt, rs, immediate
ADDIU 001001		rs		rt				immediate				addiu rt, rs, immediate
SLTI 001010		rs		rt				immediate				slti rt, rs, immediate
SLTIU 001011		rs		rt				immediate				sltiu rt, rs, immediate

B.5 转移指令

	31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000		rs	00000	00000	00000	JR 001000		jr rs
SPECIAL 000000		rs	00000	rd	00000	JALR 001001		jalr rs或jalr rd, rs
J 000010				instr_index				j target
JAL 000011				instr_index				jal target
BEQ 000100		rs	rt		offset			beq rs, rt, offset
BEQ 000100		00000	00000		offset			b offset
BGTZ 000111		rs	00000		offset			bgtz rs, offset
BLEZ 000110		rs	00000		offset			blez rs, offset
BNE 000101		rs	rt		offset			bne rs, rt, offset
REGIMM 000001		rs	BLTZ 00000		offset			bltz rs, offset
REGIMM 000001		rs	BLTZAL 10000		offset			bltzal rs, offset
REGIMM 000001		rs	BGEZ 00001		offset			bgez rs, offset
REGIMM 000001		rs	BGEZAL 10001		offset			bgezal rs, offset
REGIMM 000001		00000	BGEZAL 10001		offset			bal offset

B.6 加载存储指令

31	26 25	21 20	16 15	0	
	LB 100000	base	rt	offset	lb rt, offset(base)
	LBU 100100	base	rt	offset	lbu rt, offset(base)
	LH 100001	base	rt	offset	lh rt, offset(base)
	LHU 100101	base	rt	offset	lhu rt, offset(base)
	LW 100011	base	rt	offset	lw rt, offset(base)
	SB 101000	base	rt	offset	sb rt, offset(base)
	SH 101001	base	rt	offset	sh rt, offset(base)
	SW 101011	base	rt	offset	sw rt, offset(base)
	LWL 100010	base	rt	offset	lwl rt, offset(base)
	LWR 100110	base	rt	offset	lwr rt, offset(base)
	SWL 101010	base	rt	offset	swl rt, offset(base)
	SWR 101110	base	rt	offset	swr rt, offset(base)
	LL 110000	base	rt	offset	ll rt, offset(base)
	SC 111000	base	rt	offset	sc rt, offset(base)

B.7 协处理器访问指令

31	26 25	21 20	16 15	11 10	3 2 0	
	COP0 010000	MT 00100	rt	rd	00000000	sel mtc0 rt, rd
	COP0 010000	MF 00000	rt	rd	00000000	sel mfc0 rt, rd

B.8 异常相关指令

31	26 25	21 20	16 15	6 5	0	
SPECIAL 000000	rs	rt	code	TEQ 110100		teq rs, rt
SPECIAL 000000	rs	rt	code	TGE 110000		tge rs, rt
SPECIAL 000000	rs	rt	code	TGEU 110001		tgeu rs, rt
SPECIAL 000000	rs	rt	code	TLT 110010		tlt rs, rt
SPECIAL 000000	rs	rt	code	TLTU 110011		tltu rs, rt
SPECIAL 000000	rs	rt	code	TNE 110110		tne rs, rt
REGIMM 000001	rs	TEQI 01100		immediate		teqi rs, immediate
REGIMM 000001	rs	TGEI 01000		immediate		tgei rs, immediate
REGIMM 000001	rs	TGEIU 01001		immediate		tgeiu rs, immediate
REGIMM 000001	rs	TLTI 01010		immediate		tlti rs, immediate
REGIMM 000001	rs	TLTIU 01011		immediate		tltiu rs, immediate
REGIMM 000001	rs	TNEI 01110		immediate		tnei rs, immediate
SPECIAL 000000		code		SYSCALL 001100		syscall
COP0 010000	CO 1	0000 0000 0000 0000 000		ERET 011000		eret

B.9 空指令及其他指令

31	26 25	21 20	16 15	11 10	6 5	0	
SPECIAL 000000	00000	00000	00000	00000	SLL 000000		nop
SPECIAL 000000	00000	00000	00000	00001	SLL 000000		ssnop
SPECIAL 000000	00000	00000	00000	00001	SYNC 001111		sync
PREF 110011	base	hint		offset			pref

参考文献

- [1] 张晨曦，王志英，张春元，戴葵，肖晓强. 计算机体系结构(第2版) [M]. 北京：高等教育出版社，2005
- [2] 水头一寿，米泽辽，藤田裕士. 赵谦译. CPU自制入门 [M]. 北京：人民邮电出版社，2014
- [3] 万木杨. 大话处理器 [M]. 北京：清华大学出版社，2011
- [4] 李亚民. 计算机原理与设计——Verilog HDL版 [M]. 北京：清华大学出版社，2011
- [5] 王金明. 数字系统设计与Verilog HDL (第4版) [M]. 北京：电子工业出版社，2011
- [6] 刘佩林，谭志明，刘嘉龑. MIPS体系结构与编程 [M]. 北京：科学出版社，2008
- [7] Dominic Sweetman. 李鹏，鲍峥，石洋译. MIPS体系结构透视 [M]. 北京：机械工业出版社，2008
- [8] WISHBONE System-on-Chip(SoC) Interconnection Architecture for Portable IP Cores Revision B.3. 2002.9
- [9] 徐敏. 开源软核处理器OpenRisc的SOPC设计 [M]. 北京：北京航空航天大学出版社，2008

- [10] Jean J. Labrosse. 邵贝贝译. 嵌入式实时操作系统μC/OS-II
(第2版) [M]. 北京: 北京航空航天大学出版社, 2003
- [11] 田泽. 嵌入式系统开发与应用 [M]. 北京: 北京航空航天大学出版社, 2005